

Imbalanced Classification with Python

Choose Better Metrics, Balance
Skewed Classes, and Apply
Cost-Sensitive Learning

Jason Brownlee

MACHINE
LEARNING
MASTERY



Disclaimer

The information contained within this eBook is strictly for educational purposes. If you wish to apply ideas contained in this eBook, you are taking full responsibility for your actions.

The author has made every effort to ensure the accuracy of the information within this book was correct at time of publication. The author does not assume and hereby disclaims any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from accident, negligence, or any other cause.

No part of this eBook may be reproduced or transmitted in any form or by any means, electronic or mechanical, recording or by any information storage and retrieval system, without written permission from the author.

Acknowledgements

Special thanks to my copy editor Sarah Martin and my technical editors Michael Sanderson and Arun Koshy, Andrei Cheremskoy, and John Halfyard.

Copyright

Imbalanced Classification with Python

© Copyright 2021 Jason Brownlee. All Rights Reserved.

Edition: v1.3

Contents

Copyright	i
Contents	ii
Preface	iii
I Introduction	iv
II Foundation	1
1 What is Imbalanced Classification	2
1.1 Tutorial Overview	2
1.2 Classification Predictive Modeling	3
1.3 Imbalanced Classification Problems	4
1.4 Causes of Class Imbalance	5
1.5 Challenge of Imbalanced Classification	5
1.6 Examples of Imbalanced Classification	7
1.7 Further Reading	8
1.8 Summary	9
2 Intuition for Imbalanced Classification	10
2.1 Tutorial Overview	10
2.2 Create and Plot a Binary Classification Problem	10
2.3 Create Synthetic Dataset with a Class Distribution	12
2.4 Effect of Skewed Class Distributions	15
2.5 Further Reading	22
2.6 Summary	23
3 Challenge of Imbalanced Classification	24
3.1 Tutorial Overview	24
3.2 Why Imbalanced Classification Is Hard	25
3.3 Compounding Effect of Dataset Size	26
3.4 Compounding Effect of Label Noise	28
3.5 Compounding Effect of Data Distribution	31
3.6 Further Reading	34

3.7 Summary	34
III Model Evaluation	35
4 Tour of Model Evaluation Metrics	36
4.1 Tutorial Overview	36
4.2 Challenge of Evaluation Metrics	36
4.3 Taxonomy of Classifier Evaluation Metrics	38
4.4 How to Choose an Evaluation Metric	45
4.5 Further Reading	46
4.6 Summary	47
5 The Failure of Accuracy	48
5.1 Tutorial Overview	48
5.2 What Is Classification Accuracy?	49
5.3 Accuracy Fails for Imbalanced Classification	50
5.4 Example of Accuracy for Imbalanced Classification	51
5.5 Further Reading	55
5.6 Summary	56
6 Precision, Recall, and F-measure	57
6.1 Tutorial Overview	57
6.2 Precision Measure	58
6.3 Recall Measure	60
6.4 Precision vs. Recall	63
6.5 F-measure	64
6.6 Further Reading	66
6.7 Summary	67
7 ROC Curves and Precision-Recall Curves	68
7.1 Tutorial Overview	68
7.2 ROC Curves and ROC AUC	69
7.3 Precision-Recall Curves and AUC	74
7.4 ROC and PR Curves With a Severe Imbalance	77
7.5 Further Reading	84
7.6 Summary	85
8 Probability Scoring Methods	86
8.1 Tutorial Overview	86
8.2 Probability Metrics	86
8.3 Log Loss Score	88
8.4 Brier Score	91
8.5 Further Reading	94
8.6 Summary	95

9 Cross-Validation for Imbalanced Datasets	96
9.1 Tutorial Overview	96
9.2 Challenge of Evaluating Classifiers	97
9.3 Failure of k -Fold Cross-Validation	97
9.4 Fix Cross-Validation for Imbalanced Classification	100
9.5 Further Reading	101
9.6 Summary	102
IV Data Sampling	103
10 Tour of Data Sampling Methods	104
10.1 Tutorial Overview	104
10.2 Problem of an Imbalanced Class Distribution	105
10.3 Balance the Class Distribution With Sampling	105
10.4 Tour of Popular Data Sampling Methods	107
10.5 Further Reading	110
10.6 Summary	110
11 Random Data Sampling	112
11.1 Tutorial Overview	112
11.2 Random Sampling	113
11.3 Random Oversampling	113
11.4 Random Undersampling	117
11.5 Further Reading	119
11.6 Summary	120
12 Oversampling Methods	121
12.1 Tutorial Overview	121
12.2 Synthetic Minority Oversampling Technique	122
12.3 SMOTE for Balancing Data	123
12.4 SMOTE for Classification	126
12.5 SMOTE With Selective Sample Generation	130
12.6 Further Reading	136
12.7 Summary	137
13 Undersampling Methods	139
13.1 Tutorial Overview	139
13.2 Undersampling for Imbalanced Classification	140
13.3 Methods that Select Examples to Keep	142
13.4 Methods that Select Examples to Delete	150
13.5 Combinations of Keep and Delete Methods	155
13.6 Further Reading	160
13.7 Summary	162

14 Oversampling and Undersampling	163
14.1 Tutorial Overview	163
14.2 Binary Test Problem and Decision Tree Model	164
14.3 Manually Combine Data Sampling Methods	166
14.4 Standard Combined Data Sampling Methods	170
14.5 Further Reading	174
14.6 Summary	175
V Cost-Sensitive	177
15 Cost-Sensitive Learning	178
15.1 Tutorial Overview	178
15.2 Not All Classification Errors Are Equal	179
15.3 Cost-Sensitive Learning	180
15.4 Cost-Sensitive Imbalanced Classification	182
15.5 Cost-Sensitive Methods	184
15.6 Further Reading	187
15.7 Summary	188
16 Cost-Sensitive Logistic Regression	189
16.1 Tutorial Overview	189
16.2 Imbalanced Classification Dataset	190
16.3 Logistic Regression for Imbalanced Classification	192
16.4 Weighted Logistic Regression with Scikit-Learn	193
16.5 Grid Search Weighted Logistic Regression	197
16.6 Further Reading	199
16.7 Summary	199
17 Cost-Sensitive Decision Trees	201
17.1 Tutorial Overview	201
17.2 Imbalanced Classification Dataset	202
17.3 Decision Trees for Imbalanced Classification	205
17.4 Weighted Decision Tree With Scikit-Learn	206
17.5 Grid Search Weighted Decision Tree	207
17.6 Further Reading	209
17.7 Summary	210
18 Cost-Sensitive Support Vector Machines	211
18.1 Tutorial Overview	211
18.2 Imbalanced Classification Dataset	212
18.3 SVM for Imbalanced Classification	215
18.4 Weighted SVM With Scikit-Learn	217
18.5 Grid Search Weighted SVM	218
18.6 Further Reading	220
18.7 Summary	221

19 Cost-Sensitive Deep Learning in Keras	223
19.1 Tutorial Overview	223
19.2 Imbalanced Classification Dataset	224
19.3 Neural Network Model in Keras	225
19.4 Deep Learning for Imbalanced Classification	228
19.5 Weighted Neural Network With Keras	229
19.6 Further Reading	231
19.7 Summary	232
20 Cost-Sensitive Gradient Boosting with XGBoost	233
20.1 Tutorial Overview	233
20.2 Imbalanced Classification Dataset	234
20.3 XGBoost Model for Classification	235
20.4 Weighted XGBoost for Class Imbalance	237
20.5 Tune the Class Weighting Hyperparameter	240
20.6 Further Reading	242
20.7 Summary	243
VI Advanced Algorithms	244
21 Probability Threshold Moving	245
21.1 Tutorial Overview	245
21.2 Converting Probabilities to Class Labels	246
21.3 Threshold-Moving for Imbalanced Classification	246
21.4 Optimal Threshold for ROC Curve	248
21.5 Optimal Threshold for Precision-Recall Curve	254
21.6 Optimal Threshold Tuning	258
21.7 Further Reading	261
21.8 Summary	262
22 Probability Calibration	263
22.1 Tutorial Overview	263
22.2 Problem of Uncalibrated Probabilities	264
22.3 How to Calibrate Probabilities	265
22.4 SVM With Calibrated Probabilities	266
22.5 Decision Tree With Calibrated Probabilities	270
22.6 Grid Search Probability Calibration With KNN	272
22.7 Further Reading	275
22.8 Summary	276
23 Ensemble Algorithms	277
23.1 Tutorial Overview	277
23.2 Bagging for Imbalanced Classification	278
23.3 Random Forest for Imbalanced Classification	281
23.4 Easy Ensemble for Imbalanced Classification	285
23.5 Further Reading	287

23.6 Summary	288
24 One-Class Classification	289
24.1 Tutorial Overview	289
24.2 One-Class Classification for Imbalanced Data	290
24.3 One-Class Support Vector Machines	292
24.4 Isolation Forest	295
24.5 Minimum Covariance Determinant	296
24.6 Local Outlier Factor	298
24.7 Further Reading	300
24.8 Summary	302
VII Projects	303
25 Framework for Imbalanced Classification Projects	304
25.1 Tutorial Overview	304
25.2 What Algorithm To Use?	305
25.3 Use a Systematic Framework	305
25.4 Detailed Framework for Imbalanced Classification	306
25.5 Further Reading	318
25.6 Summary	318
26 Project: Haberman Breast Cancer Classification	319
26.1 Tutorial Overview	319
26.2 Haberman Breast Cancer Survival Dataset	320
26.3 Explore the Dataset	320
26.4 Model Test and Baseline Result	324
26.5 Evaluate Probabilistic Models	328
26.6 Make Prediction on New Data	339
26.7 Further Reading	340
26.8 Summary	341
27 Project: Oil Spill Classification	342
27.1 Tutorial Overview	342
27.2 Oil Spill Dataset	343
27.3 Explore the Dataset	344
27.4 Model Test and Baseline Result	346
27.5 Evaluate Models	349
27.6 Make Prediction on New Data	361
27.7 Further Reading	364
27.8 Summary	365
28 Project: German Credit Classification	366
28.1 Tutorial Overview	366
28.2 German Credit Dataset	367
28.3 Explore the Dataset	368

28.4 Model Test and Baseline Result	371
28.5 Evaluate Models	376
28.6 Make Prediction on New Data	385
28.7 Further Reading	388
28.8 Summary	389
29 Project: Microcalcification Classification	390
29.1 Tutorial Overview	390
29.2 Mammography Dataset	391
29.3 Explore the Dataset	392
29.4 Model Test and Baseline Result	396
29.5 Evaluate Models	399
29.6 Make Predictions on New Data	407
29.7 Further Reading	409
29.8 Summary	410
30 Project: Phoneme Classification	411
30.1 Tutorial Overview	411
30.2 Phoneme Dataset	412
30.3 Explore the Dataset	412
30.4 Model Test and Baseline Result	416
30.5 Evaluate Models	419
30.6 Make Prediction on New Data	427
30.7 Further Reading	429
30.8 Summary	430
VIII Appendix	431
A Getting Help	432
A.1 Imbalanced Classification Books	432
A.2 Machine Learning Books	432
A.3 Python APIs	432
A.4 Ask Questions About Imbalanced Classification	433
A.5 How to Ask Questions	433
A.6 Contact the Author	433
B How to Setup Python on Your Workstation	434
B.1 Tutorial Overview	434
B.2 Download Anaconda	434
B.3 Install Anaconda	436
B.4 Start and Update Anaconda	438
B.5 Install the Imbalanced-Learn Library	441
B.6 Install the Deep Learning Libraries	441
B.7 Install the XGBoost Library	442
B.8 Further Reading	443
B.9 Summary	443

IX Conclusions	444
How Far You Have Come	445

Preface

Classification predictive modeling involves assigning a class label to an example. It may be one of the most studied and used areas of machine learning. Nevertheless, the majority of the models used to learn from classification data and the metrics used to evaluate those models assume that the distribution of the examples across the class labels is equal. The field is focused on the simplest form of classification problems, so-called balanced classification problems.

When the examples across classes are imbalanced, many machine learning algorithms fail and metrics used to evaluate those models, such as classification accuracy, become dangerously misleading. Many problems have a skew in the class distribution. For example, numerous tasks where practitioners want to apply machine learning are examples of imbalanced classification, including fraud detection, churn prediction, medical diagnosis, and many more. In fact, it may be more common to have imbalanced classes than balanced classes.

Thankfully, there is a small but rapidly growing field of study dedicated to the problem of imbalanced classification. This field includes modifications to existing algorithms to make them useful for imbalanced classification, careful selection of performance metrics, and entirely new data preparation techniques and modeling algorithms. This book was carefully designed to help you bring the tools and techniques of imbalanced classification to your next project. The tutorials were designed to teach you these techniques the fastest and most effective way that I know how: to learn by doing, with executable code that you can run to develop the intuitions required and that you can copy-and-paste into your project and immediately get a result.

Imbalanced classification is important to machine learning, and I believe that if it is taught at the right level for practitioners, it can be a fascinating, fun, directly applicable, and immeasurably useful toolbox of techniques. I hope that you agree.

Jason Brownlee
2021

Part I

Introduction

Welcome

Welcome to *Imbalanced Classification with Python*. Classification predictive modeling is the task of assigning a label to an example. Imbalanced classification is those classification tasks where the distribution of examples across the classes is not equal. Typically the class distribution is severely skewed so that for each example in the minority class, there may be one hundred or even one thousand examples in the majority class. Practical imbalanced classification requires the use of a suite of specialized techniques, data preparation techniques, learning algorithms, and performance metrics. I designed this book to teach you the techniques for imbalanced classification step-by-step with concrete and executable examples in Python.

Who Is This Book For?

Before we get started, let's make sure you are in the right place. This book is for developers that may know some applied machine learning. Maybe you know how to work through a predictive modeling problem end-to-end, or at least most of the main steps, with popular tools. The lessons in this book do assume a few things about you, such as:

- You know your way around basic Python for programming.
- You may know some basic NumPy for array manipulation.
- You may know some basic scikit-learn for modeling.

This guide was written in the top-down and results-first machine learning style that you're used to from Machine Learning Mastery.

About Your Outcomes

This book will teach you the techniques for imbalanced classification that you need to know as a machine learning practitioner. After reading and working through this book, you will know:

- The challenge and intuitions for imbalanced classification datasets.
- How to choose an appropriate performance metric for evaluating models for imbalanced classification.
- How to appropriately stratify an imbalanced dataset when splitting into train and test sets and when using k -fold cross-validation.

- How to use data sampling algorithms like SMOTE to transform the training dataset for an imbalanced dataset when fitting a range of standard machine learning models.
- How algorithms from the field of cost-sensitive learning can be used for imbalanced classification.
- How to use modified versions of standard algorithms like SVM and decision trees to take the class weighting into account.
- How to tune the threshold when interpreting predicted probabilities as class labels.
- How to calibrate probabilities predicted by nonlinear algorithms that are not fit using a probabilistic framework.
- How to use algorithms from the field of outlier detection and anomaly detection for imbalanced classification.
- How to use modified ensemble algorithms that have been modified to take the class distribution into account during training.
- How to systematically work through an imbalanced classification predictive modeling project.

This book is not a substitute for an undergraduate course in imbalanced classification (if such courses exist) or a textbook for such a course, although it could complement such materials. For a good list of top papers, textbooks, and other resources on imbalanced classification, see the *Further Reading* section at the end of each tutorial.

How to Read This Book

This book was written to be read linearly, from start to finish. That being said, if you know the basics and need help with a specific method or type of problem, then you can flip straight to that section and get started. This book was designed for you to read on your workstation, on the screen, not on a tablet or eReader. My hope is that you have the book open right next to your editor and run the examples as you read about them.

This book is not intended to be read passively or be placed in a folder as a reference text. It is a playbook, a workbook, and a guidebook intended for you to learn by doing and then apply your new understanding with working Python examples. To get the most out of the book, I would recommend playing with the examples in each tutorial. Extend them, break them, then fix them.

About the Book Structure

This book was designed around major imbalanced classification techniques that are directly relevant to real-world problems. There are a lot of things you could learn about imbalanced classification, from theory to abstract concepts to APIs. My goal is to take you straight to developing an intuition for the elements you must understand with laser-focused tutorials.

The tutorials were designed to focus on how to get results with imbalanced classification methods. As such, the tutorials give you the tools to both rapidly understand and apply each technique or operation. There is a mixture of both tutorial lessons and projects to both introduce the methods and give plenty of examples and opportunities to practice using them.

Each of the tutorials is designed to take you about one hour to read through and complete, excluding the extensions and further reading. You can choose to work through the lessons one per day, one per week, or at your own pace. I think momentum is critically important, and this book is intended to be read and used, not to sit idle. I recommend picking a schedule and sticking to it. The tutorials are divided into six parts; they are:

- **Part 1: Foundations.** Discover a gentle introduction to the field of imbalanced classification, the intuitions for skewed class distributions, and properties of datasets that make these problems challenging.
- **Part 2: Model Evaluation.** Discover the failure of classification accuracy for skewed class distributions and alternate performance metrics such as precision-recall, area under ROC curves, and probability scoring methods.
- **Part 3: Data Sampling.** Discover techniques for transforming the training dataset to balance the class distribution, including data oversampling, undersampling, and combinations of these techniques.
- **Part 4: Cost-Sensitive.** Discover modified versions of machine learning algorithms that allow different types of misclassification errors to have a different cost on model performance.
- **Part 5: Advanced Algorithms.** Discover advanced algorithms for interpreting and calibrating predicted probabilities for imbalanced classification, as well as the use of ensemble algorithms and techniques from the field of anomaly detection.
- **Part 6: Projects.** Discover how to put the techniques from imbalanced classification into practice with end-to-end projects on real datasets that have skewed class distributions.

Each part targets a specific learning outcome, and so does each tutorial within each part. This acts as a filter to ensure you are only focused on the things you need to know to get to a specific result and do not get bogged down in the math or near-infinite number of digressions.

The tutorials were not designed to teach you everything there is to know about each of the methods. They were designed to give you an understanding of how they work, how to use them, and how to interpret the results the fastest way I know how: to learn by doing.

About Python Code Examples

The code examples were carefully designed to demonstrate the purpose of a given lesson. For this reason, the examples are highly targeted.

- Algorithms were demonstrated on synthetic datasets to give you the context and confidence to bring the techniques to your own classification datasets.

- Model configurations used were discovered through trial and error and are skillful, but not optimized. This leaves the door open for you to explore new and possibly better configurations.
- Code examples are complete and standalone. The code for each lesson will run as-is with no code from prior lessons or third parties needed beyond the installation of the required packages.

A complete working example is presented with each tutorial for you to inspect and copy-paste. All source code is also provided with the book and I would recommend running the provided files whenever possible to avoid any copy-paste issues. The provided code was developed in a text editor and intended to be run on the command line. No special IDE or notebooks are required. If you are using a more advanced development environment and are having trouble, try running the example from the command line instead.

Machine learning algorithms are stochastic. This means that they will make different predictions when the same model configuration is trained on the same training data. On top of that, each experimental problem in this book is based around generating stochastic predictions. As a result, this means you will not get exactly the same sample output presented in this book. This is by design. I want you to get used to the stochastic nature of the machine learning algorithms. If this bothers you, please note:

- You can re-run a given example a few times and your results should be close to the values reported.
- You can make the output consistent by fixing the random number seed.
- You can develop a robust estimate of the skill of a model by fitting and evaluating it multiple times and taking the average of the final skill score (highly recommended).

All code examples were tested on a POSIX-compatible machine with Python 3. All code examples will run on modest and modern computer hardware. I am only human, and there may be a bug in the sample code. If you discover a bug, please let me know so I can fix it and correct the book (and you can request a free update any time).

About Further Reading

Each lesson includes a list of further reading resources. This may include:

- Research papers.
- Books and book chapters.
- Webpages.
- API documentation.
- Open source projects.

Wherever possible, I have listed and linked to the relevant API documentation for key objects and functions used in each lesson so you can learn more about them. When it comes to research papers, I have listed those that are first to use a specific technique or first in a specific problem domain. These are not required reading but can give you more technical details, theory, and configuration details if you're looking for it. Wherever possible, I have tried to link to the freely available version of the paper on arxiv.org. You can search for and download any of the papers listed on Google Scholar Search scholar.google.com. Wherever possible, I have tried to link to books on Amazon.

I don't know everything, and if you discover a good resource related to a given lesson, please let me know so I can update the book.

About Getting Help

You might need help along the way. Don't worry; you are not alone.

- **Help with a technique?** If you need help with the technical aspects of a specific operation or technique, see the *Further Reading* section at the end of each tutorial.
- **Help with APIs?** If you need help with using a Python library, see the list of resources in the *Further Reading* section at the end of each lesson, and also see *Appendix A*.
- **Help with your workstation?** If you need help setting up your environment, I would recommend using Anaconda and following my tutorial in *Appendix B*.
- **Help in general?** You can shoot me an email. My details are in *Appendix A*.

Next

Are you ready? Let's dive in! Next up you will discover the importance of imbalanced classification problems.

Part II

Foundation

Chapter 1

What is Imbalanced Classification

Classification predictive modeling involves predicting a class label for a given observation. An imbalanced classification problem is an example of a classification problem where the distribution of examples across the known classes is not equal. The distribution can vary from a slight bias to a severe imbalance where there is one example in the minority class for hundreds, thousands, or millions of examples in the majority class or classes.

Imbalanced classifications pose a challenge for predictive modeling as most of the machine learning algorithms used for classification were designed around the assumption of an equal number of examples for each class. This results in models that have poor predictive performance, specifically for the minority class. This is a problem because typically, the minority class is more important and therefore the problem is more sensitive to classification errors for the minority class than the majority class. In this tutorial, you will discover imbalanced classification predictive modeling. After completing this tutorial, you will know:

- Imbalanced classification is the problem of classification when there is an unequal distribution of classes in the training dataset.
- The imbalance in the class distribution may vary, but a severe imbalance is more challenging to model and may require specialized techniques.
- Many real-world classification problems have an imbalanced class distribution, such as fraud detection, spam detection, and churn prediction.

Let's get started.

1.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Classification Predictive Modeling
2. Imbalanced Classification Problems
3. Causes of Class Imbalance
4. Challenge of Imbalanced Classification
5. Examples of Imbalanced Classification

1.2 Classification Predictive Modeling

Classification is a predictive modeling problem that involves assigning a class label to each observation.

... classification models generate a predicted class, which comes in the form of a discrete category. For most practical applications, a discrete category prediction is required in order to make a decision.

— Page 248, *Applied Predictive Modeling*, 2013.

Each example is comprised of both the observations and a class label.

- **Example:** An observation from the domain (input) and an associated class label (output).

For example, we may collect measurements of a flower and classify the species of flower (label) from the measurements. The number of classes for a predictive modeling problem is typically fixed when the problem is framed or described, and typically, the number of classes does not change. We may alternately choose to predict a probability of class membership instead of a crisp class label. This allows a predictive model to share uncertainty in a prediction across a range of options and allows the user to interpret the result in the context of the problem.

Like regression models, classification models produce a continuous valued prediction, which is usually in the form of a probability (i.e., the predicted values of class membership for any individual sample are between 0 and 1 and sum to 1).

— Page 248, *Applied Predictive Modeling*, 2013.

Using the previous example, given measurements of a flower (observation), we may predict the likelihood (probability) of the flower being an example of each of twenty different species of flower. The number of classes for a predictive modeling problem is typically fixed when the problem is framed or described, and usually, the number of classes does not change.

A classification predictive modeling problem may have two class labels. This is the simplest type of classification problem and is referred to as two-class classification or binary classification. Alternately, the problem may have more than two classes, such as three, 10, or even hundreds of classes. These types of problems are referred to as multiclass classification problems.

- **Binary Classification Problem:** A classification predictive modeling problem where all examples belong to one of two classes.
- **Multiclass Classification Problem:** A classification predictive modeling problem where all examples belong to one of three or more classes.

When working on classification predictive modeling problems, we must collect a training dataset. A training dataset is a number of examples from the domain that include both the input data (e.g. measurements) and the output data (e.g. class label).

- **Training Dataset:** A number of examples collected from the problem domain that include the input observations and output class labels.

Depending on the complexity of the problem and the types of models we may choose to use, we may need tens, hundreds, thousands, or even millions of examples from the domain to constitute a training dataset. The training dataset is used to better understand the input data to help best prepare it for modeling. It is also used to evaluate a suite of different modeling algorithms. It is used to tune the hyperparameters of a chosen model. And finally, the training dataset is used to train a final model on all available data that we can use in the future to make predictions for new examples from the problem domain.

Now that we are familiar with classification predictive modeling, let's consider an imbalance of classes in the training dataset.

1.3 Imbalanced Classification Problems

The number of examples that belong to each class may be referred to as the class distribution. Imbalanced classification refers to a classification predictive modeling problem where the number of examples in the training dataset for each class label is not balanced. Specifically, where the class distribution is not equal or close to equal, and is instead biased or skewed.

- **Imbalanced Classification:** A classification predictive modeling problem where the distribution of examples across the classes is not equal.

For example, we may collect measurements of flowers and have 80 examples of one flower species and 20 examples of a second flower species, and only these examples comprise our training dataset. This represents an example of an imbalanced classification problem.

An imbalance occurs when one or more classes have very low proportions in the training data as compared to the other classes.

— Page 419, *Applied Predictive Modeling*, 2013.

We refer to these types of problems as *imbalanced classification* instead of *unbalanced classification*. Unbalanced refers to a class distribution that was balanced and is now no longer balanced, whereas imbalanced refers to a class distribution that is inherently not balanced. There are other less general names that may be used to describe these types of classification problems, such as:

- Rare event prediction.
- Extreme event prediction.
- Severe class imbalance.

The imbalance of a problem is defined by the distribution of classes in a specific training dataset.

... class imbalance must be defined with respect to a particular dataset or distribution. Since class labels are required in order to determine the degree of class imbalance, class imbalance is typically gauged with respect to the training distribution.

— Page 16, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

It is common to describe the imbalance of classes in a dataset in terms of a ratio. For example, an imbalanced binary classification problem with an imbalance of 1 to 100 (1:100) means that for every one example in one class, there are 100 examples in the other class. Another way to describe the imbalance of classes in a dataset is to summarize the class distribution as percentages of the training dataset. For example, an imbalanced multiclass classification problem may have 80 percent examples in the first class, 18 percent in the second class, and 2 percent in a third class.

Now that we are familiar with the definition of an imbalanced classification problem, let's look at some possible reasons as to why the classes may be imbalanced.

1.4 Causes of Class Imbalance

The imbalance of the class distribution in an imbalanced classification predictive modeling problem may have many causes. There are perhaps two main groups of causes for the imbalance we may want to consider; they are data sampling and properties of the domain. It is possible that the imbalance in the examples across the classes was caused by the way the examples were collected or sampled from the problem domain. This might involve biases introduced during data collection, and errors made during data collection.

- Biased Sampling.
- Measurement Errors.

For example, perhaps examples were collected from a narrow geographical region, or slice of time, and the distribution of classes may be quite different or perhaps even collected in a different way. Errors may have been made when collecting the observations. One type of error might have been applying the wrong class labels to many examples. Alternately, the processes or systems from which examples were collected may have been damaged or impaired to cause the imbalance.

Often in cases where the imbalance is caused by a sampling bias or measurement error, the imbalance can be corrected by improved sampling methods, and/or correcting the measurement error. This is because the training dataset is not a fair representation of the problem domain that is being addressed.

The imbalance might be a property of the problem domain. For example, the natural occurrence or presence of one class may dominate other classes. This may be because the process that generates observations in one class is more expensive in time, cost, computation, or other resources. As such, it is often infeasible or intractable to simply collect more samples from the domain in order to improve the class distribution. Instead, a model is required to learn the difference between the classes. Now that we are familiar with the possible causes of a class imbalance, let's consider why imbalanced classification problems are challenging.

1.5 Challenge of Imbalanced Classification

The imbalance of the class distribution will vary across problems. A classification problem may be a little skewed, such as if there is a slight imbalance. Alternately, the classification problem

may have a severe imbalance where there might be hundreds or thousands of examples in one class and tens of examples in another class for a given training dataset.

- **Slight Imbalance.** An imbalanced classification problem where the distribution of examples is uneven by a small amount in the training dataset (e.g. 4:6).
- **Severe Imbalance.** An imbalanced classification problem where the distribution of examples is uneven by a large amount in the training dataset (e.g. 1:100 or more).

Most of the contemporary works in class imbalance concentrate on imbalance ratios ranging from 1:4 up to 1:100. [...] In real-life applications such as fraud detection or cheminformatics we may deal with problems with imbalance ratio ranging from 1:1000 up to 1:5000.

— *Learning From Imbalanced Data: Open Challenges And Future Directions*, 2016.

A slight imbalance is often not a concern, and the problem can often be treated like a normal classification predictive modeling problem. A severe imbalance of the classes can be challenging to model and may require the use of specialized techniques.

Any dataset with an unequal class distribution is technically imbalanced. However, a dataset is said to be imbalanced when there is a significant, or in some cases extreme, disproportion among the number of examples of each class of the problem.

— Page 19, *Learning from Imbalanced Data Sets*, 2018.

The class or classes with abundant examples are called the major or majority classes, whereas the class with few examples (and there is typically just one) is called the minor or minority class.

- **Majority Class:** The class (or classes) in an imbalanced classification predictive modeling problem that has more examples.
- **Minority Class:** The class in an imbalanced classification predictive modeling problem that has less examples.

When working with an imbalanced classification problem, the minority class is typically of the most interest. This means that a model's skill in correctly predicting the class label or probability for the minority class is more important than the majority class or classes.

Developments in learning from imbalanced data have been mainly motivated by numerous real-life applications in which we face the problem of uneven data representation. In such cases the minority class is usually the more important one and hence we require methods to improve its recognition rates.

— *Learning From Imbalanced Data: Open Challenges And Future Directions*, 2016.

The minority class is harder to predict because there are few examples of this class, by definition. This means it is more challenging for a model to learn the characteristics of examples from this class, and to differentiate examples from this class from the majority class (or classes).

The abundance of examples from the majority class (or classes) can swamp the minority class. Most machine learning algorithms for classification predictive models are designed and demonstrated on problems that assume an equal distribution of classes. This means that a naive application of a model may focus on learning the characteristics of the abundant observations only, neglecting the examples from the minority class that is, in fact, of more interest and whose predictions are more valuable.

... the learning process of most classification algorithms is often biased toward the majority class examples, so that minority ones are not well modeled into the final system.

— Page vii, *Learning from Imbalanced Data Sets*, 2018.

Imbalanced classification is not *solved*. It remains an open problem generally, and practically must be identified and addressed specifically for each training dataset. This is true even in the face of more data, so-called *big data*, large neural network models, so-called *deep learning*, and very impressive competition-winning models, so-called *xgboost*.

Despite intense works on imbalanced learning over the last two decades there are still many shortcomings in existing methods and problems yet to be properly addressed.

— *Learning From Imbalanced Data: Open Challenges And Future Directions*, 2016.

Now that we are familiar with the challenge of imbalanced classification, let's look at some common examples.

1.6 Examples of Imbalanced Classification

Many of the classification predictive modeling problems that we are interested in solving in practice are imbalanced. As such, it is surprising that imbalanced classification does not get more attention than it does.

Imbalanced learning not only presents significant new challenges to the data research community but also raises many critical questions in real-world data- intensive applications, ranging from civilian applications such as financial and biomedical data analysis to security- and defense-related applications such as surveillance and military data analysis.

— Page 2, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

Below is a list of ten examples of problem domains where the class distribution of examples is inherently imbalanced. Many classification problems may have a severe imbalance in the class distribution; nevertheless, looking at common problem domains that are inherently imbalanced will make the ideas and challenges of class imbalance concrete.

- Fraud Detection.
- Claim Prediction
- Default Prediction.
- Churn Prediction.
- Spam Detection.
- Anomaly Detection.
- Outlier Detection.
- Intrusion Detection
- Conversion Prediction.

The list of examples sheds light on the nature of imbalanced classification predictive modeling. Each of these problem domains represents an entire field of study, where specific problems from each domain can be framed and explored as imbalanced classification predictive modeling. This highlights the multidisciplinary nature of class imbalanced classification, and why it is so important for a machine learning practitioner to be aware of the problem and skilled in addressing it.

Imbalance can be present in any data set or application, and hence, the practitioner should be aware of the implications of modeling this type of data.

— Page 419, *Applied Predictive Modeling*, 2013.

Notice that most, if not all, of the examples are likely binary classification problems. Notice too that examples from the minority class are rare, extreme, abnormal, or unusual in some way. Also notice that many of the domains are described as *detection*, highlighting the desire to discover the minority class amongst the abundant examples of the majority class. We now have a robust overview of imbalanced classification predictive modeling.

1.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

1.7.1 Papers

- *Learning From Imbalanced Data: Open Challenges And Future Directions*, 2016.
<https://link.springer.com/article/10.1007/s13748-016-0094-0>

1.7.2 Books

- Chapter 16: Remedies for Severe Class Imbalance, *Applied Predictive Modeling*, 2013.
<https://amzn.to/32M80ta>
- *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.
<https://amzn.to/32K9K6d>
- *Learning from Imbalanced Data Sets*, 2018.
<https://amzn.to/307Xlva>

1.7.3 Articles

- Anomaly detection, Wikipedia.
https://en.wikipedia.org/wiki/Anomaly_detection

1.8 Summary

In this tutorial, you discovered imbalanced classification predictive modeling. Specifically, you learned:

- Imbalanced classification is the problem of classification when there is an unequal distribution of classes in the training dataset.
- The imbalance in the class distribution may vary, but a severe imbalance is more challenging to model and may require specialized techniques.
- Many real-world classification problems have an imbalanced class distribution such as fraud detection, spam detection, and churn prediction.

1.8.1 Next

In the next tutorial, you will discover how to develop an intuition for different skews in the class distribution.

Chapter 2

Intuition for Imbalanced Classification

An imbalanced classification problem is a problem that involves predicting a class label where the distribution of class labels in the training dataset is not equal. A challenge for beginners working with imbalanced classification problems is what a specific skewed class distribution means. For example, what is the difference and implication for a 1:10 vs. a 1:100 class ratio?

Differences in the class distribution for an imbalanced classification problem will influence the choice of data preparation and modeling algorithms. Therefore it is critical that practitioners develop an intuition for the implications for different class distributions. In this tutorial, you will discover how to develop a practical intuition for imbalanced and highly skewed class distributions. After completing this tutorial, you will know:

- How to create a synthetic dataset for binary classification and plot the examples by class.
- How to create synthetic classification datasets with any given class distribution.
- How different skewed class distributions actually look in practice.

Let's get started.

2.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Create and Plot a Binary Classification Problem
2. Create Synthetic Dataset With Class Distribution
3. Effect of Skewed Class Distributions

2.2 Create and Plot a Binary Classification Problem

The scikit-learn Python machine learning library provides functions for generating synthetic datasets. The `make_blobs()` function can be used to generate a specified number of examples from a test classification problem with a specified number of classes. The function returns the input and output parts of each example ready for modeling. For example, the snippet below will generate 1,000 examples for a two-class (binary) classification problem with two input variables. The class values have the values of 0 and 1.

```
...
X, y = make_blobs(n_samples=1000, centers=2, n_features=2, random_state=1, cluster_std=3)
```

Listing 2.1: Example of creating a synthetic dataset.

Once generated, we can then plot the dataset to get an intuition for the spatial relationship between the examples. Because there are only two input variables, we can create a scatter plot to plot each example as a point. This can be achieved with the `scatter()` Matplotlib function.

The color of the points can then be varied based on the class values. This can be achieved by first selecting the array indexes for the examples for a given class, then only plotting those points, then repeating the select-and-plot process for the other class. The `where()` NumPy function can be used to retrieve the array indexes that match a criterion, such as a class label having a given value. For example:

```
...
# create scatter plot for samples from each class
for class_value in range(2):
    # get row indexes for samples with this class
    row_ix = where(y == class_value)
    # create scatter of these samples
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
```

Listing 2.2: Example of plotting data points by class label.

Tying this together, the complete example of creating a binary classification test dataset and plotting the examples as a scatter plot is listed below.

```
# generate binary classification dataset and plot
from numpy import where
from matplotlib import pyplot
from sklearn.datasets import make_blobs
# generate dataset
X, y = make_blobs(n_samples=1000, centers=2, n_features=2, random_state=1, cluster_std=3)
# create scatter plot for samples from each class
for class_value in range(2):
    # get row indexes for samples with this class
    row_ix = where(y == class_value)
    # create scatter of these samples
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
# show the plot
pyplot.show()
```

Listing 2.3: Example of creating and plotting a balanced dataset.

Running the example creates the dataset and scatter plot, showing the examples for each of the two classes with different colors. We can see that there is an equal number of examples in each class, in this case, 500, and that we can imagine drawing a line to reasonably separate the classes, much like a classification predictive model might in learning how to discriminate the examples.

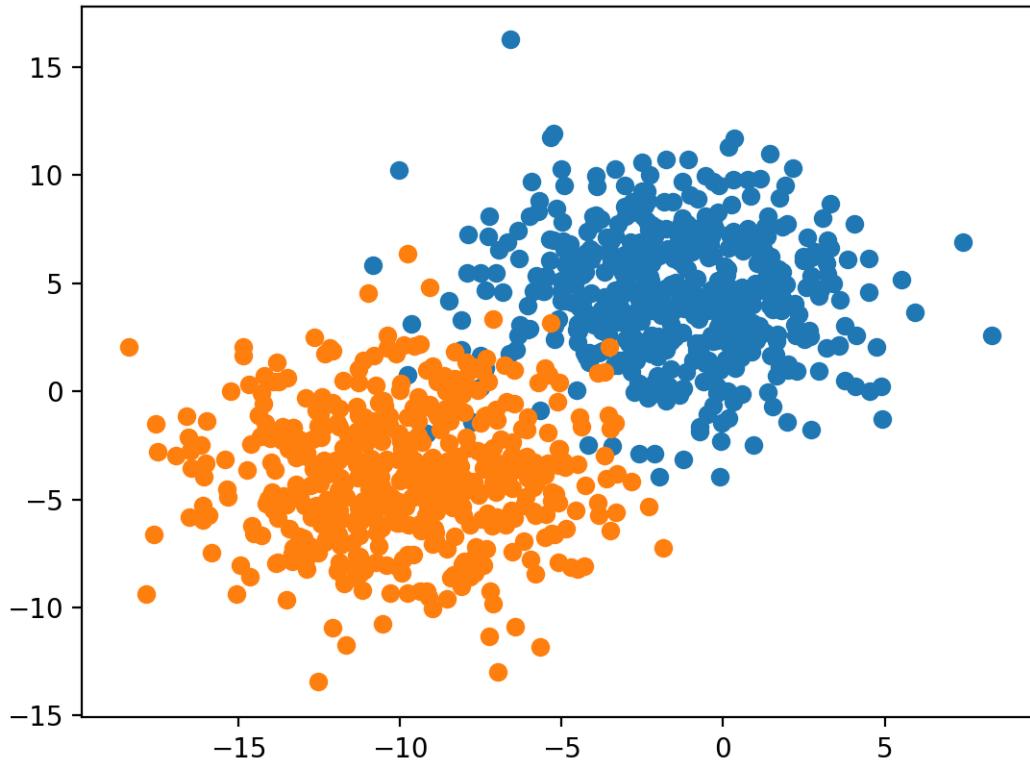


Figure 2.1: Scatter Plot of Binary Classification Dataset.

Now that we know how to create a synthetic binary classification dataset and plot the examples, let's look at the example of class imbalances on the example.

2.3 Create Synthetic Dataset with a Class Distribution

The `make_blobs()` function will always create synthetic datasets with an equal class distribution. Nevertheless, we can use this function to create synthetic classification datasets with arbitrary class distributions with a few extra lines of code. A class distribution can be defined as a dictionary where the key is the class value (e.g. 0 or 1) and the value is the number of randomly generated examples to include in the dataset. For example, an equal class distribution with 5,000 examples in each class would be defined as:

```
...
# define the class distribution
proportions = {0:5000, 1:5000}
```

Listing 2.4: Example of defining the class distribution.

We can then enumerate through the different distributions and find the largest distribution, then use the `make_blobs()` function to create a dataset with that many examples for each of the classes.

```

...
# determine the number of classes
n_classes = len(proportions)
# determine the number of examples to generate for each class
largest = max([v for k,v in proportions.items()])
n_samples = largest * n_classes

```

Listing 2.5: Example of locating the largest distribution.

This is a good starting point, but will give us more samples than are required for each class label. We can then enumerate through the class labels and select the desired number of examples for each class to comprise the dataset that will be returned.

```

...
# collect the examples
X_list, y_list = list(), list()
for k,v in proportions.items():
    row_ix = where(y == k)[0]
    selected = row_ix[:v]
    X_list.append(X[selected, :])
    y_list.append(y[selected])

```

Listing 2.6: Example of selecting examples for each class label according to the proportions.

We can tie this together into a new function named `get_dataset()` that will take a class distribution and return a synthetic dataset with that class distribution.

```

# create a dataset with a given class distribution
def get_dataset(proportions):
    # determine the number of classes
    n_classes = len(proportions)
    # determine the number of examples to generate for each class
    largest = max([v for k,v in proportions.items()])
    n_samples = largest * n_classes
    # create dataset
    X, y = make_blobs(n_samples=n_samples, centers=n_classes, n_features=2, random_state=1,
                       cluster_std=3)
    # collect the examples
    X_list, y_list = list(), list()
    for k,v in proportions.items():
        row_ix = where(y == k)[0]
        selected = row_ix[:v]
        X_list.append(X[selected, :])
        y_list.append(y[selected])
    return vstack(X_list), hstack(y_list)

```

Listing 2.7: Example of a function for creating an imbalanced classification dataset.

The function can take any number of classes, although we will use it for simple binary classification problems. Next, we can take the code from the previous section for creating a scatter plot for a created dataset and place it in a helper function. Below is the `plot_dataset()` function that will plot the dataset and show a legend to indicate the mapping of colors to class labels.

```

# scatter plot of dataset, different color for each class
def plot_dataset(X, y):

```

```
# create scatter plot for samples from each class
n_classes = len(unique(y))
for class_value in range(n_classes):
    # get row indexes for samples with this class
    row_ix = where(y == class_value)[0]
    # create scatter of these samples
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(class_value))
# show a legend
pyplot.legend()
# show the plot
pyplot.show()
```

Listing 2.8: Example of a function for plotting a dataset colored by class label.

Finally, we can test these new functions. We will define a dataset with 5,000 examples for each class (10,000 total examples), and plot the result. The complete example is listed below.

```
# create and plot synthetic dataset with a given class distribution
from numpy import unique
from numpy import hstack
from numpy import vstack
from numpy import where
from matplotlib import pyplot
from sklearn.datasets import make_blobs

# create a dataset with a given class distribution
def get_dataset(proportions):
    # determine the number of classes
    n_classes = len(proportions)
    # determine the number of examples to generate for each class
    largest = max([v for k,v in proportions.items()])
    n_samples = largest * n_classes
    # create dataset
    X, y = make_blobs(n_samples=n_samples, centers=n_classes, n_features=2, random_state=1,
                       cluster_std=3)
    # collect the examples
    X_list, y_list = list(), list()
    for k,v in proportions.items():
        row_ix = where(y == k)[0]
        selected = row_ix[:v]
        X_list.append(X[selected, :])
        y_list.append(y[selected])
    return vstack(X_list), hstack(y_list)

# scatter plot of dataset, different color for each class
def plot_dataset(X, y):
    # create scatter plot for samples from each class
    n_classes = len(unique(y))
    for class_value in range(n_classes):
        # get row indexes for samples with this class
        row_ix = where(y == class_value)[0]
        # create scatter of these samples
        pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(class_value))
    # show a legend
    pyplot.legend()
    # show the plot
    pyplot.show()
```

```
# define the class distribution
proportions = {0:5000, 1:5000}
# generate dataset
X, y = get_dataset(proportions)
# plot dataset
plot_dataset(X, y)
```

Listing 2.9: Example of creating and plotting a dataset with configurable class balance.

Running the example creates the dataset and plots the result as before, although this time with our provided class distribution. In this case, we have many more examples for each class and a helpful legend to indicate the mapping of plot colors to class labels.

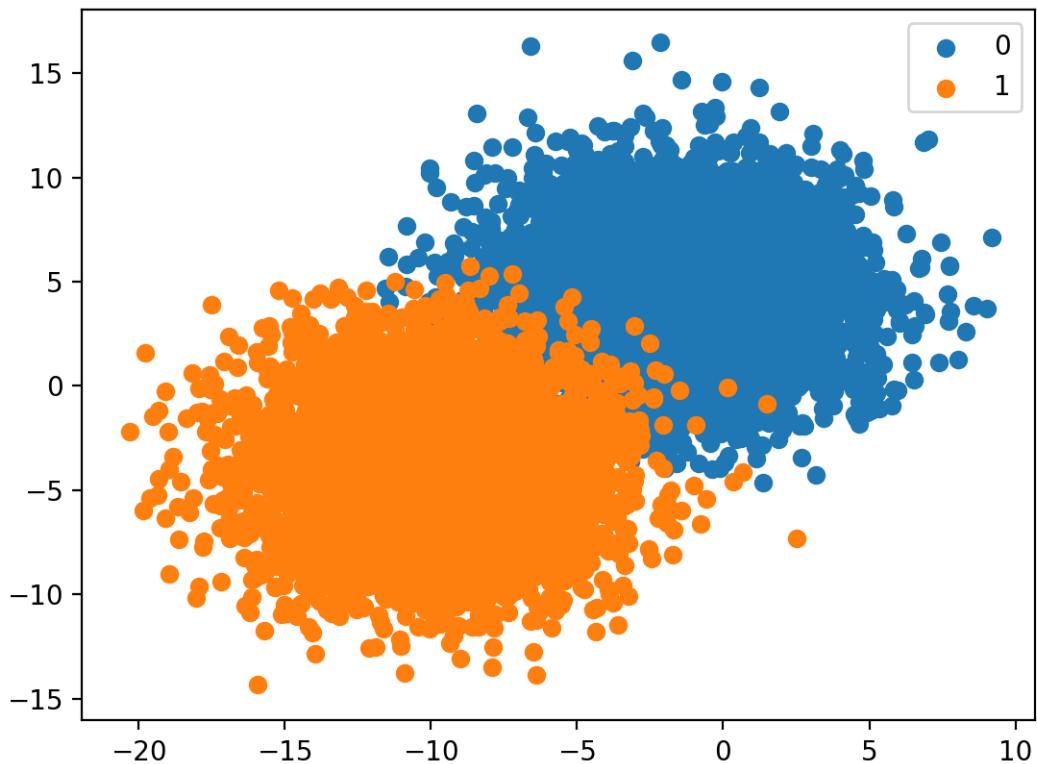


Figure 2.2: Scatter Plot of Binary Classification Dataset With Provided Class Distribution.

Now that we have the tools to create and plot a synthetic dataset with arbitrary skewed class distributions, let's look at the effect of different distributions.

2.4 Effect of Skewed Class Distributions

It is important to develop an intuition for the spatial relationship for different class imbalances. For example, what is the 1:1000 class distribution relationship like? It is an abstract relationship

and we need to tie it to something concrete. We can generate synthetic test datasets with different imbalanced class distributions and use that as a basis for developing an intuition for different skewed distributions we might be likely to encounter in real datasets.

Reviewing scatter plots of different class distributions can give a rough feeling for the relationship between the classes that can be useful when thinking about the selection of techniques and evaluation of models when working with similar class distributions in the future. They provide a point of reference. We have already seen a 1:1 relationship in the previous section (e.g. 5000:5000).

Note that when working with binary classification problems, especially imbalanced problems, it is important that the majority class is assigned to class 0 and the minority class is assigned to class 1. This is because many evaluation metrics will assume this relationship. Therefore, we can ensure our class distributions meet this practice by defining the majority then the minority classes in the call to the `get_dataset()` function; for example:

```
...
# define the class distribution
proportions = {0:10000, 1:10}
# generate dataset
X, y = get_dataset(proportions)
...
```

Listing 2.10: Example of configuring the class distribution.

In this section, we can look at different skewed class distributions with the size of the minority class increasing on a log scale, such as:

- 1:10
- 1:100
- 1:1000

Let's take a closer look at each class distribution in turn.

2.4.1 1:10 Imbalanced Class Distribution

A 1:10 class distribution with 10,000 to 1,000 examples means that there will be 11,000 examples in the dataset, with about 91 percent for class 0 and about 9 percent for class 1. The complete code example is listed below.

```
# create and plot synthetic dataset with a given class distribution
from numpy import unique
from numpy import hstack
from numpy import vstack
from numpy import where
from matplotlib import pyplot
from sklearn.datasets import make_blobs

# create a dataset with a given class distribution
def get_dataset(proportions):
    # determine the number of classes
    n_classes = len(proportions)
    # determine the number of examples to generate for each class
```

```

largest = max([v for k,v in proportions.items()])
n_samples = largest * n_classes
# create dataset
X, y = make_blobs(n_samples=n_samples, centers=n_classes, n_features=2, random_state=1,
    cluster_std=3)
# collect the examples
X_list, y_list = list(), list()
for k,v in proportions.items():
    row_ix = where(y == k)[0]
    selected = row_ix[:v]
    X_list.append(X[selected, :])
    y_list.append(y[selected])
return vstack(X_list), hstack(y_list)

# scatter plot of dataset, different color for each class
def plot_dataset(X, y):
    # create scatter plot for samples from each class
    n_classes = len(unique(y))
    for class_value in range(n_classes):
        # get row indexes for samples with this class
        row_ix = where(y == class_value)[0]
        # create scatter of these samples
        pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(class_value))
    # show a legend
    pyplot.legend()
    # show the plot
    pyplot.show()

# define the class distribution
proportions = {0:10000, 1:1000}
# generate dataset
X, y = get_dataset(proportions)
# plot dataset
plot_dataset(X, y)

```

Listing 2.11: Example of creating and plotting a dataset with a 1:10 class balance.

Running the example creates the dataset with the defined class distribution and plots the result. Although the balance seems stark, the plot shows that about 10 percent of the points in the minority class compared to the majority class is not as bad as we might think. The relationship appears manageable, although if the classes overlapped significantly, we can imagine a very different story.

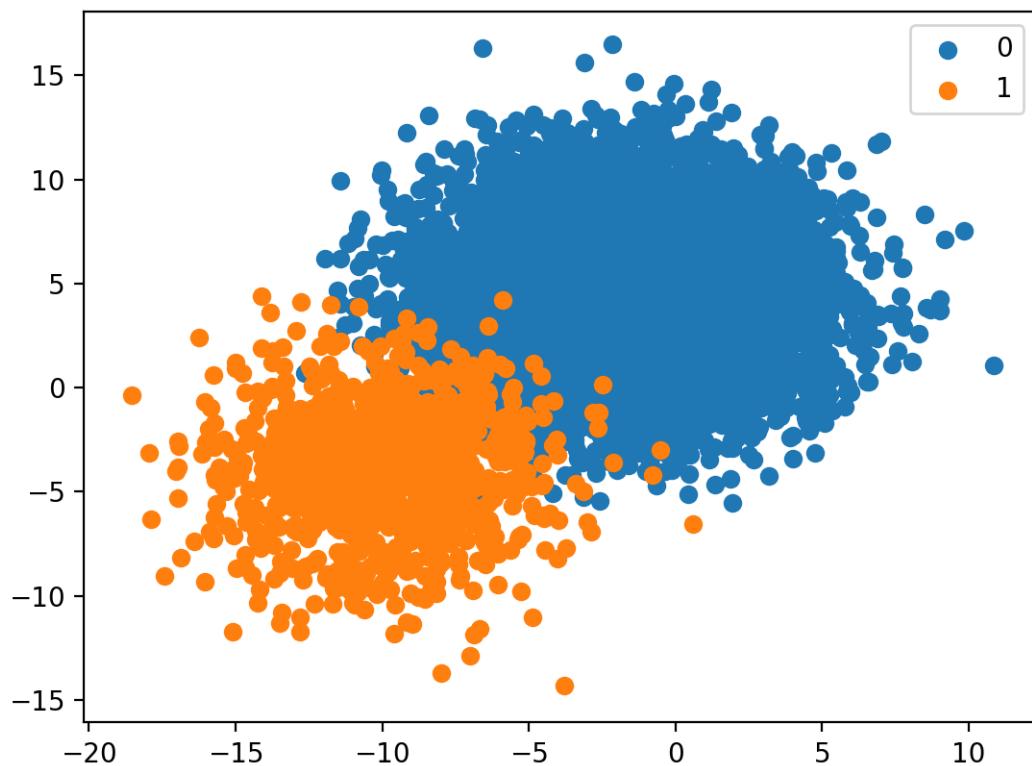


Figure 2.3: Scatter Plot of Binary Classification Dataset With a 1 to 10 Class Distribution.

2.4.2 1:100 Imbalanced Class Distribution

A 1:100 class distribution with 10,000 to 100 examples means that there will be 10,100 examples in the dataset, with about 99 percent for class 0 and about 1 percent for class 1. The complete code example is listed below.

```
# create and plot synthetic dataset with a given class distribution
from numpy import unique
from numpy import hstack
from numpy import vstack
from numpy import where
from matplotlib import pyplot
from sklearn.datasets import make_blobs

# create a dataset with a given class distribution
def get_dataset(proportions):
    # determine the number of classes
    n_classes = len(proportions)
    # determine the number of examples to generate for each class
    largest = max([v for k,v in proportions.items()])
    n_samples = largest * n_classes
    # create dataset
```

```

X, y = make_blobs(n_samples=n_samples, centers=n_classes, n_features=2, random_state=1,
                  cluster_std=3)
# collect the examples
X_list, y_list = list(), list()
for k,v in proportions.items():
    row_ix = where(y == k)[0]
    selected = row_ix[:v]
    X_list.append(X[selected, :])
    y_list.append(y[selected])
return vstack(X_list), hstack(y_list)

# scatter plot of dataset, different color for each class
def plot_dataset(X, y):
    # create scatter plot for samples from each class
    n_classes = len(unique(y))
    for class_value in range(n_classes):
        # get row indexes for samples with this class
        row_ix = where(y == class_value)[0]
        # create scatter of these samples
        pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(class_value))
    # show a legend
    pyplot.legend()
    # show the plot
    pyplot.show()

# define the class distribution
proportions = {0:10000, 1:100}
# generate dataset
X, y = get_dataset(proportions)
# plot dataset
plot_dataset(X, y)

```

Listing 2.12: Example of creating and plotting a dataset with a 1:100 class balance.

Running the example creates the dataset with the defined class distribution and plots the result. A 1 to 100 relationship is a large skew. The plot makes this clear with what feels like a sprinkling of points compared to the enormous mass of the majority class. It is most likely that a real-world dataset will fall somewhere on the line between a 1:10 and 1:100 class distribution and the plot for 1:100 really highlights the need to carefully consider each point in the minority class, both in terms of measurement errors (e.g. outliers) and in terms of prediction errors that might be made by a model.

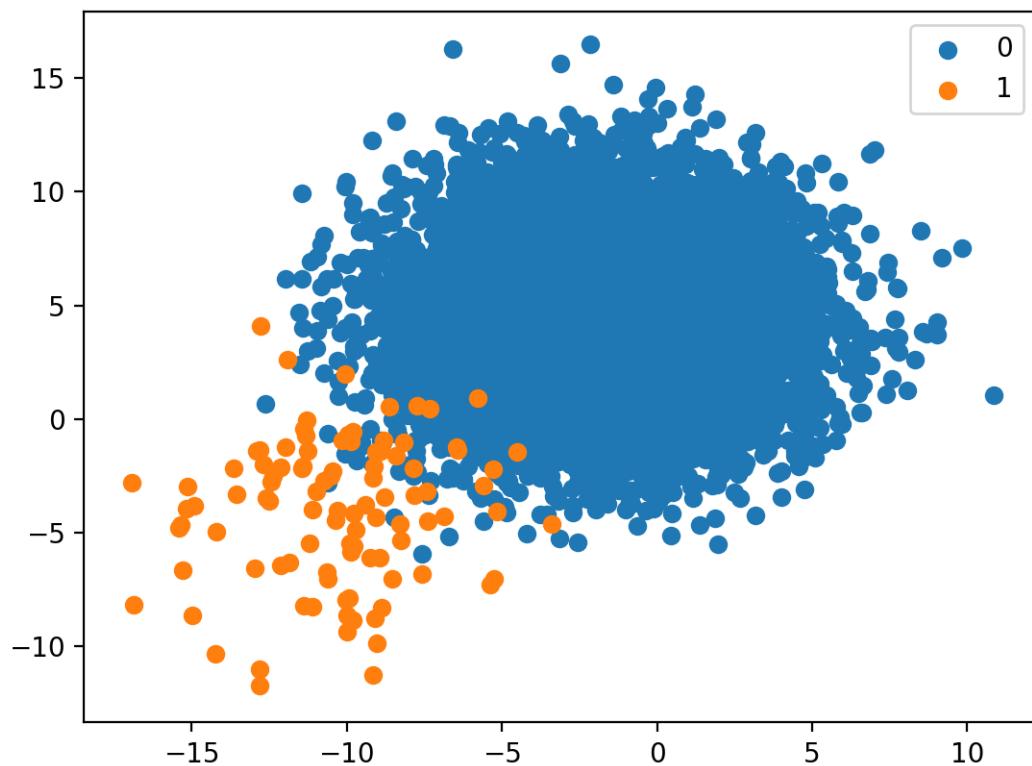


Figure 2.4: Scatter Plot of Binary Classification Dataset With a 1 to 100 Class Distribution.

2.4.3 1:1000 Imbalanced Class Distribution

A 1:100 class distribution with 10,000 to 10 examples means that there will be 10,010 examples in the dataset, with about 99.9 percent for class 0 and about 0.1 percent for class 1. The complete code example is listed below.

```
# create and plot synthetic dataset with a given class distribution
from numpy import unique
from numpy import hstack
from numpy import vstack
from numpy import where
from matplotlib import pyplot
from sklearn.datasets import make_blobs

# create a dataset with a given class distribution
def get_dataset(proportions):
    # determine the number of classes
    n_classes = len(proportions)
    # determine the number of examples to generate for each class
    largest = max([v for k,v in proportions.items()])
    n_samples = largest * n_classes
    # create dataset
```

```

X, y = make_blobs(n_samples=n_samples, centers=n_classes, n_features=2, random_state=1,
                  cluster_std=3)
# collect the examples
X_list, y_list = list(), list()
for k,v in proportions.items():
    row_ix = where(y == k)[0]
    selected = row_ix[:v]
    X_list.append(X[selected, :])
    y_list.append(y[selected])
return vstack(X_list), hstack(y_list)

# scatter plot of dataset, different color for each class
def plot_dataset(X, y):
    # create scatter plot for samples from each class
    n_classes = len(unique(y))
    for class_value in range(n_classes):
        # get row indexes for samples with this class
        row_ix = where(y == class_value)[0]
        # create scatter of these samples
        pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(class_value))
    # show a legend
    pyplot.legend()
    # show the plot
    pyplot.show()

# define the class distribution
proportions = {0:10000, 1:10}
# generate dataset
X, y = get_dataset(proportions)
# plot dataset
plot_dataset(X, y)

```

Listing 2.13: Example of creating and plotting a dataset with a 1:1000 class balance.

Running the example creates the dataset with the defined class distribution and plots the result. As we might already suspect, a 1 to 1,000 relationship is aggressive. In our chosen setup, just 10 examples of the minority class are present to 10,000 of the majority class. With such a lack of data, we can see that on modeling problems with such a dramatic skew, that we should probably spend a lot of time on the actual minority examples that are available and see if domain knowledge can be used in some way. Automatic modeling methods will have a tough challenge.

This example also highlights another important aspect orthogonal to the class distribution and that is the number of examples. For example, although the dataset has a 1:1000 class distribution, having only 10 examples of the minority class is very challenging. Although, if we had the same class distribution with 1,000,000 of the majority class and 1,000 examples of the minority class, the additional 990 minority class examples would likely be invaluable in developing an effective model.

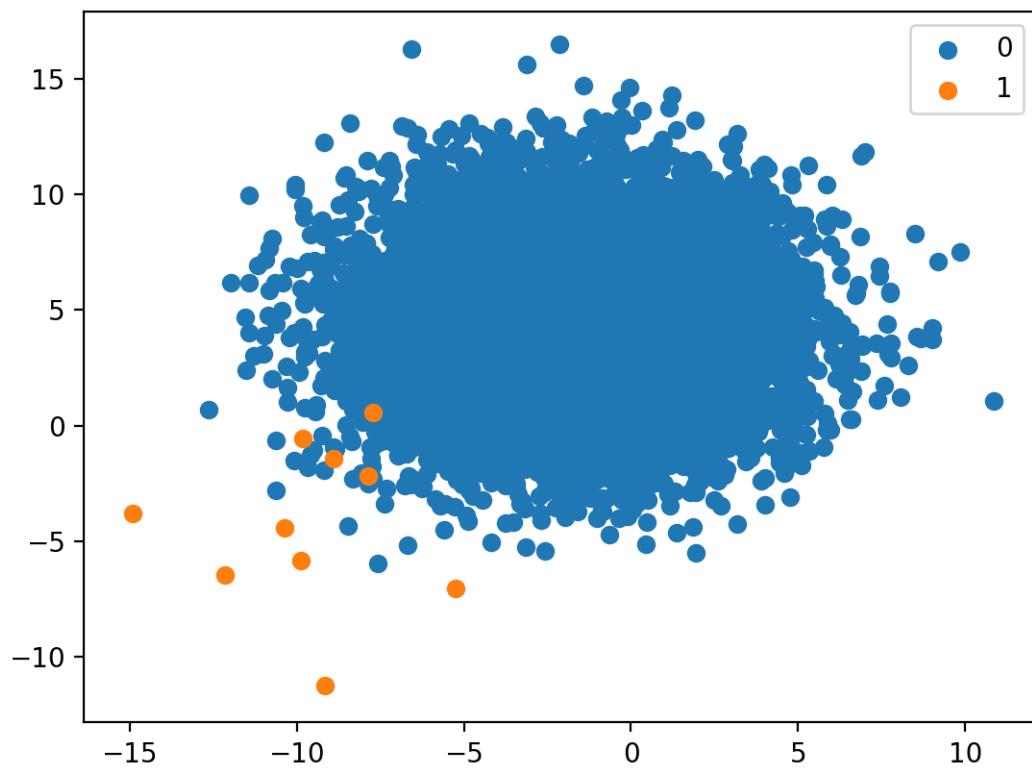


Figure 2.5: Scatter Plot of Binary Classification Dataset With a 1 to 1000 Class Distribution.

2.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

2.5.1 API

- `sklearn.datasets.make_blobs` API.
https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html
- `matplotlib.pyplot.scatter` API.
https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.scatter.html
- `numpy.where` API.
<https://docs.scipy.org/doc/numpy/reference/generated/numpy.where.html>

2.6 Summary

In this tutorial, you discovered how to develop a practical intuition for imbalanced and highly skewed class distributions. Specifically, you learned:

- How to create a synthetic dataset for binary classification and plot the examples by class.
- How to create synthetic classification datasets with any given class distribution.
- How different skewed class distributions actually look in practice.

2.6.1 Next

In the next tutorial, you will discover the properties of a classification task that make imbalanced classification more challenging.

Chapter 3

Challenge of Imbalanced Classification

Imbalanced classification is primarily challenging as a predictive modeling task because of the severely skewed class distribution. This is the cause for poor performance with traditional machine learning models and evaluation metrics that assume a balanced class distribution. Nevertheless, there are additional properties of a classification dataset that are not only challenging for predictive modeling but also increase or compound the difficulty when modeling imbalanced datasets. In this tutorial, you will discover data characteristics that compound the challenge of imbalanced classification. After completing this tutorial, you will know:

- Imbalanced classification is specifically hard because of the severely skewed class distribution and the unequal misclassification costs.
- The difficulty of imbalanced classification is compounded by properties such as dataset size, label noise, and data distribution.
- How to develop an intuition for the compounding effects on modeling difficulty posed by different dataset properties.

Let's get started.

3.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Why Imbalanced Classification Is Hard
2. Compounding Effect of Dataset Size
3. Compounding Effect of Label Noise
4. Compounding Effect of Data Distribution

3.2 Why Imbalanced Classification Is Hard

Imbalanced classification is defined by a dataset with a skewed class distribution. This is often exemplified by a binary (two-class) classification task where most of the examples belong to class 0 with only a few examples in class 1. The distribution may range in severity from 1:2, 1:10, 1:100, or even 1:1000.

Because the class distribution is not balanced, most machine learning algorithms will perform poorly and require modification to avoid simply predicting the majority class in all cases. Additionally, metrics like classification accuracy lose their meaning and alternate methods for evaluating predictions on imbalanced examples are required, like ROC area under curve. This is the foundational challenge of imbalanced classification.

An additional level of complexity comes from the problem domain from which the examples were drawn. It is common for the majority class to represent a normal case in the domain, whereas the minority class represents an abnormal case, such as a fault, fraud, outlier, anomaly, disease state, and so on. As such, the interpretation of misclassification errors may differ across the classes.

For example, misclassifying an example from the majority class as an example from the minority class called a false positive is often not desired, but less critical than classifying an example from the minority class as belonging to the majority class, a so-called false negative. This is referred to as cost sensitivity of misclassification errors and is a second foundational challenge of imbalanced classification.

These two aspects, the skewed class distribution and cost sensitivity, are typically referenced when describing the difficulty of imbalanced classification. Nevertheless, there are other characteristics of the classification problem that, when combined with these properties, compound their effect. These are general characteristics of classification predictive modeling that magnify the difficulty of the imbalanced classification task.

Class imbalance was widely acknowledged as a complicating factor for classification.

However, some studies also argue that the imbalance ratio is not the only cause of performance degradation in learning from imbalanced data.

— Page 253, *Learning from Imbalanced Data Sets*, 2018.

There are many such characteristics, but perhaps three of the most common include:

- Dataset Size.
- Label Noise.
- Data Distribution.

It is important to not only acknowledge these properties but to also specifically develop an intuition for their impact. This will allow you to select and develop techniques to address them in your own predictive modeling projects.

Understanding these data intrinsic characteristics, as well as their relationship with class imbalance, is crucial for applying existing and developing new techniques to deal with imbalance data.

— Pages 253–254, *Learning from Imbalanced Data Sets*, 2018.

In the following sections, we will take a closer look at each of these properties and their impact on imbalanced classification.

3.3 Compounding Effect of Dataset Size

Dataset size simply refers to the number of examples collected from the domain to fit and evaluate a predictive model. Typically, more data is better as it provides more coverage of the domain, perhaps to a point of diminishing returns. Specifically, more data provides better representation of combinations and variance of features in the feature space and their mapping to class labels. From this, a model can better learn and generalize a class boundary to discriminate new examples in the future.

If the ratio of examples in the majority class to the minority class is somewhat fixed, then we would expect that we would have more examples in the minority class as the size of the dataset is scaled up. This is good if we can collect more examples. It is a problem typically because data is hard or expensive to collect and we often collect and work with a lot less data than we might prefer. As such, this can dramatically impact our ability to gain a large enough or representative sample of examples from the minority class.

A problem that often arises in classification is the small number of training instances. This issue, often reported as data rarity or lack of data, is related to the “lack of density” or “insufficiency of information”.

— Page 261, *Learning from Imbalanced Data Sets*, 2018.

For example, for a modest classification task with a balanced class distribution, we might be satisfied with thousands or tens of thousands of examples in order to develop, evaluate, and select a model. A balanced binary classification with 10,000 examples would have 5,000 examples of each class. An imbalanced dataset with a 1:100 distribution with the same number of examples would only have 100 examples of the minority class.

As such, the size of the dataset dramatically impacts the imbalanced classification task, and datasets that are thought large in general are, in fact, probably not large enough when working with an imbalanced classification problem.

Without a sufficient large training set, a classifier may not generalize characteristics of the data. Furthermore, the classifier could also overfit the training data, with a poor performance in out-of-sample tests instances.

— Page 261, *Learning from Imbalanced Data Sets*, 2018.

To help, let’s make this concrete with a worked example. We can use the `make_classification()` scikit-learn function to create a dataset of a given size with a ratio of about 1:100 examples (1 percent to 99 percent) in the minority class to the majority class. The class distribution is specified via the `weights` argument that takes a list of percentages that must add to one (e.g. [0.99, 0.01]). In the case of two classes, only the distribution of class 0 is required from which the distribution of class 1 will be inferred (e.g. [0.99], and 0.01 inferred).

```
...
# create the dataset
X, y = make_classification(n_samples=1000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
```

Listing 3.1: Example of defining a synthetic imbalanced binary classification dataset.

We can then create a scatter plot of the dataset and color the points for each class with a separate color to get an idea of the spatial relationship for the examples.

```
...
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
```

Listing 3.2: Example of creating a scatter plot of the dataset and coloring points by class.

This process can then be repeated with different datasets sizes to show how the class imbalance is impacted visually. We will compare datasets with 100, 1,000, 10,000, and 100,000 examples. The complete example is listed below.

```
# vary the dataset size for a 1:100 imbalanced dataset
from collections import Counter
from sklearn.datasets import make_classification
from matplotlib import pyplot
from numpy import where
# dataset sizes
sizes = [100, 1000, 10000, 100000]
# create and plot a dataset with each size
for i in range(len(sizes)):
    # determine the dataset size
    n = sizes[i]
    # create the dataset
    X, y = make_classification(n_samples=n, n_features=2, n_redundant=0,
                               n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
    # summarize class distribution
    counter = Counter(y)
    print('Size=%d, Ratio=%s' % (n, counter))
    # define subplot
    pyplot.subplot(2, 2, 1+i)
    pyplot.title('n=%d' % n)
    pyplot.xticks([])
    pyplot.yticks([])
    # scatter plot of examples by class label
    for label, _ in counter.items():
        row_ix = where(y == label)[0]
        pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
    pyplot.legend()
# show the figure
pyplot.show()
```

Listing 3.3: Example of creating differently sized datasets with the same class imbalance.

Running the example creates and plots the same dataset with a 1:100 class distribution using four different sizes. First, the class distribution is displayed for each dataset size. We can

see that with a small dataset of 100 examples, we only get one example in the minority class as we might expect. Even with 100,000 examples in the dataset, we only get 1,000 examples in the minority class.

```
Size=100, Ratio=Counter({0: 99, 1: 1})
Size=1000, Ratio=Counter({0: 990, 1: 10})
Size=10000, Ratio=Counter({0: 9900, 1: 100})
Size=100000, Ratio=Counter({0: 99000, 1: 1000})
```

Listing 3.4: Example output from creating differently sized datasets with the same class imbalance.

Scatter plots are created for each differently sized dataset. We can see that it is not until very large sample sizes that the underlying structure of the class distributions becomes obvious. These plots highlight the critical role that dataset size plays in imbalanced classification. It is hard to see how a model given 990 examples of the majority class and 10 of the minority class could hope to do well on the same problem depicted after 100,000 examples are drawn.

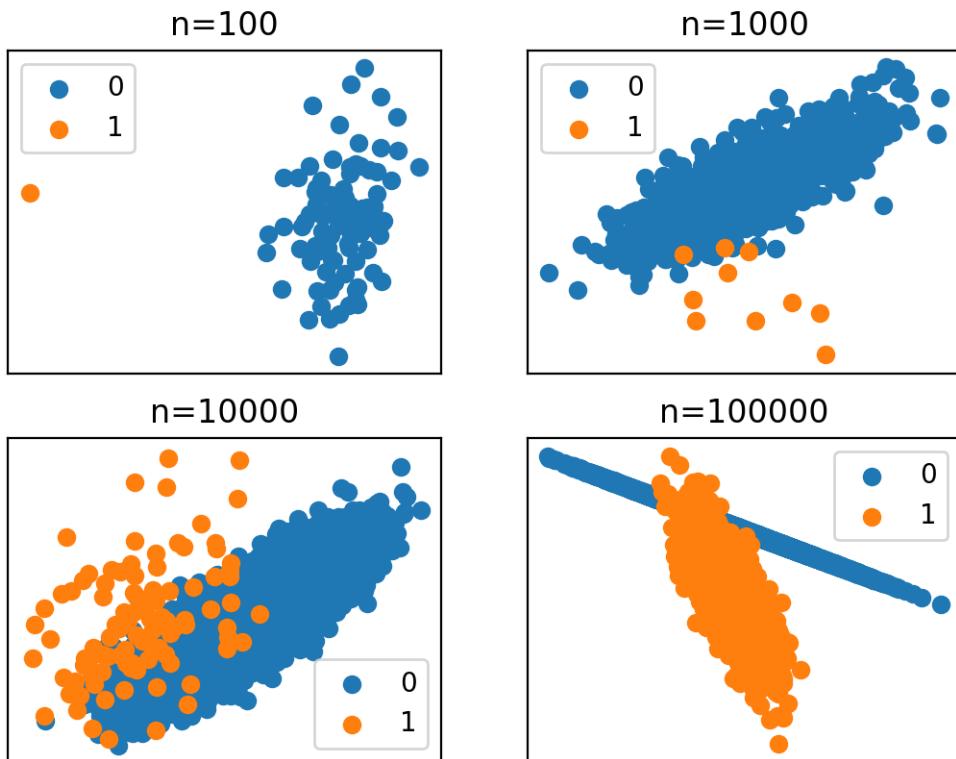


Figure 3.1: Scatter Plots of an Imbalanced Classification Dataset With Different Dataset Sizes.

3.4 Compounding Effect of Label Noise

Label noise refers to examples that belong to one class that are labeled as another class. This can make determining the class boundary in feature space problematic for most machine learning

algorithms, and this difficulty typically increases in proportion to the percentage of noise in the labels.

Two types of noise are distinguished in the literature: feature (or attribute) and class noise. Class noise is generally assumed to be more harmful than attribute noise in ML [...] class noise somehow affects the observed class values (e.g., by somehow flipping the label of a minority class instance to the majority class label).

— Page 264, *Learning from Imbalanced Data Sets*, 2018.

The cause is often inherent in the problem domain, such as ambiguous observations on the class boundary or even errors in the data collection that could impact observations anywhere in the feature space. For imbalanced classification, noisy labels have an even more dramatic effect. Given that examples in the positive class are so few, losing some to noise reduces the amount of information available about the minority class.

Additionally, having examples from the majority class incorrectly marked as belonging to the minority class can cause a disjoint or fragmentation of the minority class that is already sparse because of the lack of observations. We can imagine that if there are examples along the class boundary that are ambiguous, we could identify and remove or correct them. Examples marked for the minority class that are in areas of the feature space that are high density for the majority class are also likely easy to identify and remove or correct.

It is the case where observations for both classes are sparse in the feature space where this problem becomes particularly difficult in general, and especially for imbalanced classification. It is these situations where unmodified machine learning algorithms will define the class boundary in favor of the majority class at the expense of the minority class.

Mislabeled minority class instances will contribute to increase the perceived imbalance ratio, as well as introduce mislabeled noisy instances inside the class region of the minority class. On the other hand, mislabeled majority class instances may lead the learning algorithm, or imbalanced treatment methods, to focus on wrong areas of input space.

— Page 264, *Learning from Imbalanced Data Sets*, 2018.

We can develop an example to give a flavor of this challenge. We can hold the dataset size constant as well as the 1:100 class ratio and vary the amount of label noise. This can be achieved by setting the `flip_y` argument to the `make_classification()` function which is a percentage of the number of examples in each class to change or flip the label. We will explore varying this from 0 percent, 1 percent, 5 percent, and 7 percent. The complete example is listed below.

```
# vary the label noise for a 1:100 imbalanced dataset
from collections import Counter
from sklearn.datasets import make_classification
from matplotlib import pyplot
from numpy import where
# label noise ratios
noise = [0, 0.01, 0.05, 0.07]
# create and plot a dataset with different label noise
for i in range(len(noise)):
    # determine the label noise
```

```

n = noise[i]
# create the dataset
X, y = make_classification(n_samples=1000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=n, random_state=1)
# summarize class distribution
counter = Counter(y)
print('Noise=%d%%, Ratio=%s' % (int(n*100), counter))
# define subplot
pyplot.subplot(2, 2, 1+i)
pyplot.title('noise=%d%%' % int(n*100))
pyplot.xticks([])
pyplot.yticks([])
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
# show the figure
pyplot.show()

```

Listing 3.5: Example of creating datasets with different amounts of label noise and the same class imbalance.

Running the example creates and plots the same dataset with a 1:100 class distribution using four different amounts of label noise. First, the class distribution is printed for each dataset with differing amounts of label noise. We can see that, as we might expect, as the noise is increased, the number of examples in the minority class is increased, most of which are incorrectly labeled. We might expect these additional 31 examples in the minority class with 7 percent label noise to be quite damaging to a model trying to define a crisp class boundary in the feature space.

```

Noise=0%, Ratio=Counter({0: 990, 1: 10})
Noise=1%, Ratio=Counter({0: 983, 1: 17})
Noise=5%, Ratio=Counter({0: 963, 1: 37})
Noise=7%, Ratio=Counter({0: 959, 1: 41})

```

Listing 3.6: Example output from creating datasets with different amounts of label noise and the same class imbalance.

Scatter plots are created for each dataset with the differing label noise. In this specific case, we don't see many examples of confusion on the class boundary. Instead, we can see that as the label noise is increased, the number of examples in the mass of the majority class (orange points) increases, representing false positives that really should be identified and removed from the dataset prior to modeling.

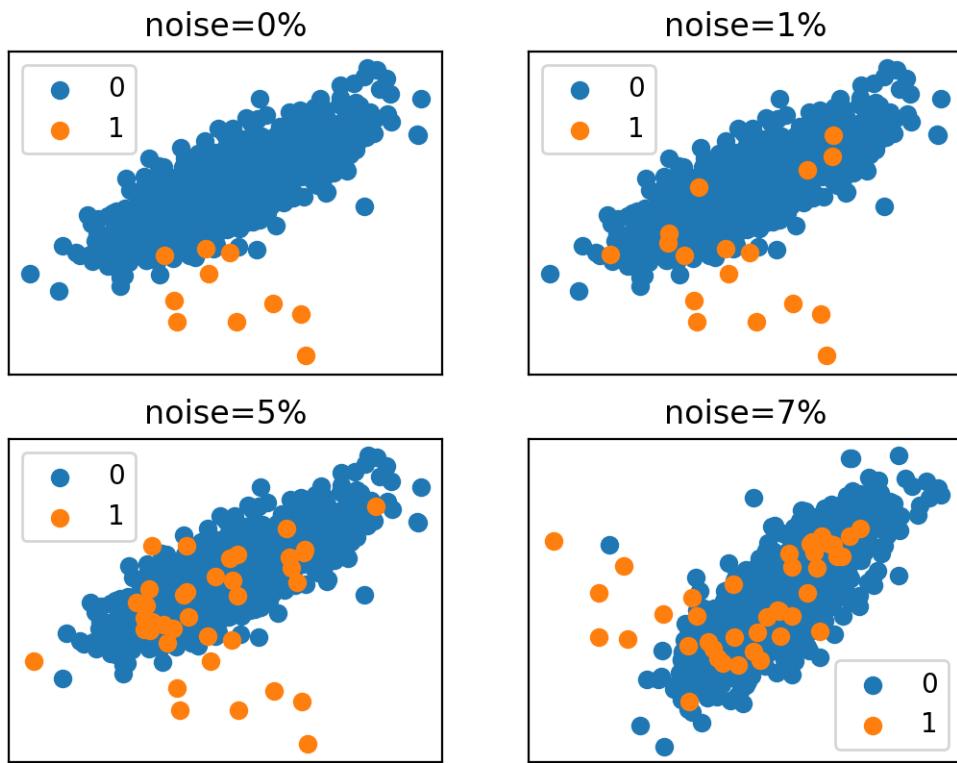


Figure 3.2: Scatter Plots of an Imbalanced Classification Dataset With Different Label Noise.

3.5 Compounding Effect of Data Distribution

Another important consideration is the distribution of examples in feature space. If we think about feature space spatially, we might like all examples in one class to be located on one part of the space, and those from the other class to appear in another part of the space. If this is the case, we have good class separability and machine learning models can draw crisp class boundaries and achieve good classification performance. This holds on datasets with a balanced or imbalanced class distribution. This is rarely the case, and it is more likely that each class has multiple *concepts* resulting in multiple different groups or clusters of examples in feature space.

... it is common that the “concept” beneath a class is split into several sub-concepts, spread over the input space.

— Page 255, *Learning from Imbalanced Data Sets*, 2018.

These groups are formally referred to as *disjuncts*, coming from a definition in the of rule-based systems for a rule that covers a group of cases comprised of sub-concepts. A small disjunct is one that relates or *covers* few examples in the training dataset.

Systems that learn from examples do not usually succeed in creating a purely conjunctive definition for each concept. Instead, they create a definition that consists of several disjuncts, where each disjunct is a conjunctive definition of a subconcept of the original concept.

— *Concept Learning And The Problem Of Small Disjuncts*, 1989.

This grouping makes class separability hard, requiring each group or cluster to be identified and included in the definition of the class boundary, implicitly or explicitly. In the case of imbalanced datasets, this is a particular problem if the minority class has multiple concepts or clusters in the feature space. This is because the density of examples in this class is already sparse and it is difficult to discern separate groupings with so few examples. It may look like one large sparse grouping.

This lack of homogeneity is particularly problematic in algorithms based on the strategy of dividing-and-conquering [...] where the sub-concepts lead to the creation of small disjuncts.

— Page 255, *Learning from Imbalanced Data Sets*, 2018.

For example, we might consider data that describes whether a patient is healthy (majority class) or sick (minority class). The data may capture many different types of illnesses, and there may be groups of similar illnesses, but if there are so few cases, then any grouping or concepts within the class may not be apparent and may look like a diffuse set mixed in with healthy cases. To make this concrete, we can look at an example.

We can use the number of clusters in the dataset as a proxy for *concepts* and compare a dataset with one cluster of examples per class to a second dataset with two clusters per class. This can be achieved by varying the `n_clusters_per_class` argument for the `make_classification()` function used to create the dataset. We would expect that in an imbalanced dataset, such as a 1:100 class distribution, that the increase in the number of clusters is obvious for the majority class, but not so for the minority class. The complete example is listed below.

```
# vary the number of clusters for a 1:100 imbalanced dataset
from collections import Counter
from sklearn.datasets import make_classification
from matplotlib import pyplot
from numpy import where
# number of clusters
clusters = [1, 2]
# create and plot a dataset with different numbers of clusters
for i in range(len(clusters)):
    c = clusters[i]
    # define dataset
    X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                               n_clusters_per_class=c, weights=[0.99], flip_y=0, random_state=1)
    counter = Counter(y)
    # define subplot
    pyplot.subplot(1, 2, 1+i)
    pyplot.title('Clusters=%d' % c)
    pyplot.xticks([])
    pyplot.yticks([])
```

```
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
# show the figure
pyplot.show()
```

Listing 3.7: Example of creating datasets with different numbers of clusters and the same class imbalance.

Running the example creates and plots the same dataset with a 1:100 class distribution using two different numbers of clusters. In the first scatter plot (left), we can see one cluster per class. The majority class (blue) quite clearly has one cluster, whereas the structure of the minority class (orange) is less obvious. In the second plot (right), we can clearly see that the majority class has two clusters, and the structure of the minority class (orange) is diffuse and it is not apparent that samples were drawn from two clusters.

This highlights the relationship between the size of the dataset and its ability to expose the underlying density or distribution of examples in the minority class. With so few examples, generalization by machine learning models is challenging, if not very problematic.

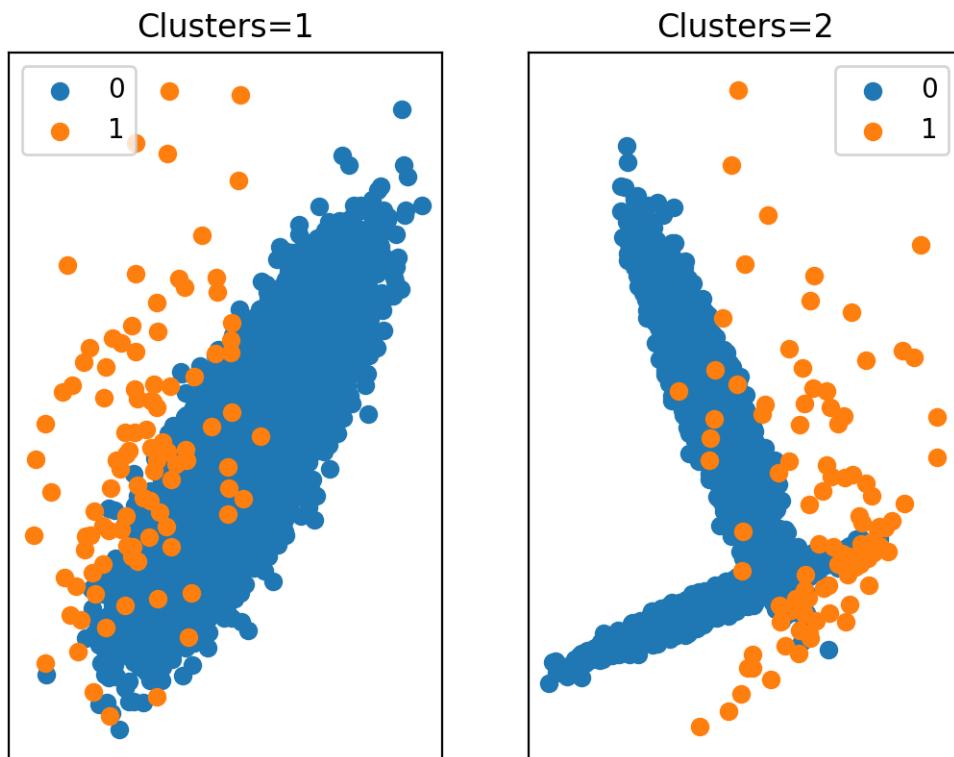


Figure 3.3: Scatter Plots of an Imbalanced Classification Dataset With Different Numbers of Clusters.

3.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

3.6.1 Papers

- *Concept Learning And The Problem Of Small Disjuncts*, 1989.
<https://dl.acm.org/citation.cfm?id=1623884>

3.6.2 Books

- *Learning from Imbalanced Data Sets*, 2018.
<https://amzn.to/307Xlva>
- *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.
<https://amzn.to/32K9K6d>

3.6.3 APIs

- `sklearn.datasets.make_classification` API.
https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification.html

3.7 Summary

In this tutorial, you discovered data characteristics that compound the challenge of imbalanced classification. Specifically, you learned:

- Imbalanced classification is specifically hard because of the severely skewed class distribution and the unequal misclassification costs.
- The difficulty of imbalanced classification is compounded by properties such as dataset size, label noise, and data distribution.
- How to develop an intuition for the compounding effects on modeling difficulty posed by different dataset properties.

3.7.1 Next

This was the final tutorial in this Part. In the next Part, you will discover the model performance metrics you can use for imbalanced classification.

Part III

Model Evaluation

Chapter 4

Tour of Model Evaluation Metrics

A classifier is only as good as the metric used to evaluate it. If you choose the wrong metric to evaluate your models, you are likely to choose a poor model, or in the worst case, be misled about the expected performance of your model.

Choosing an appropriate metric is challenging generally in applied machine learning, but is particularly difficult for imbalanced classification problems. Firstly, because most of the standard metrics that are widely used assume a balanced class distribution, and because typically not all classes, and therefore, not all prediction errors, are equal for imbalanced classification. In this tutorial, you will discover metrics that you can use for imbalanced classification. After completing this tutorial, you will know:

- About the challenge of choosing metrics for classification, and how it is particularly difficult when there is a skewed class distribution.
- How there are three main types of metrics for evaluating classifier models, referred to as rank, threshold, and probability.
- How to choose a metric for imbalanced classification if you don't know where to start.

Let's get started.

4.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Challenge of Evaluation Metrics
2. Taxonomy of Classifier Evaluation Metrics
3. How to Choose an Evaluation Metric

4.2 Challenge of Evaluation Metrics

An evaluation metric quantifies the performance of a predictive model. This typically involves training a model on a dataset, using the model to make predictions on a holdout dataset not

used during training, then comparing the predictions to the expected values in the holdout dataset.

For classification problems, metrics involve comparing the expected class label to the predicted class label or interpreting the predicted probabilities for the class labels for the problem. Selecting a model, and even the data preparation methods together are a search problem that is guided by the evaluation metric. Experiments are performed with different models and the outcome of each experiment is quantified with a metric.

Evaluation measures play a crucial role in both assessing the classification performance and guiding the classifier modeling.

— *Classification Of Imbalanced Data: A Review*, 2009.

There are standard metrics that are widely used for evaluating classification predictive models, such as classification accuracy or classification error. Standard metrics work well on most problems, which is why they are widely adopted. But all metrics make assumptions about the problem or about what is important in the problem. Therefore an evaluation metric must be chosen that best captures what you or your project stakeholders believe is important about the model or predictions, which makes choosing model evaluation metrics challenging.

This challenge is made even more difficult when there is a skew in the class distribution. The reason for this is that many of the standard metrics become unreliable or even misleading when classes are imbalanced, or severely imbalanced, such as 1:100 or 1:1000 ratio between a minority and majority class.

In the case of class imbalances, the problem is even more acute because the default, relatively robust procedures used for unskewed data can break down miserably when the data is skewed.

— Page 187, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

For example, reporting classification accuracy for a severely imbalanced classification problem could be dangerously misleading. This is the case if project stakeholders use the results to draw conclusions or plan new projects.

In fact, the use of common metrics in imbalanced domains can lead to sub-optimal classification models and might produce misleading conclusions since these measures are insensitive to skewed domains.

— *A Survey of Predictive Modelling under Imbalanced Distributions*, 2015.

Importantly, different evaluation metrics are often required when working with imbalanced classification.

Unlike standard evaluation metrics that treat all classes as equally important, imbalanced classification problems typically rate classification errors with the minority class as more important than those with the majority class. As such performance metrics may be needed that focus on the minority class, which is made challenging because it is the minority class where we lack observations required to train an effective model.

The main problem of imbalanced data sets lies on the fact that they are often associated with a user preference bias towards the performance on cases that are poorly represented in the available data sample.

— *A Survey of Predictive Modelling under Imbalanced Distributions*, 2015.

Now that we are familiar with the challenge of choosing a model evaluation metric, let's look at some examples of different metrics from which we might choose.

4.3 Taxonomy of Classifier Evaluation Metrics

There are tens of metrics to choose from when evaluating classifier models, and perhaps hundreds, if you consider all of the pet versions of metrics proposed by academics. In order to get a handle on the metrics that you could choose from, we will use a taxonomy proposed by Cesar Ferri, et al. in their 2008 paper titled *An Experimental Comparison Of Performance Measures For Classification*. It was also adopted in the 2013 book titled *Imbalanced Learning* and I think proves useful.

We can divide evaluation metrics into three useful groups; they are:

1. Threshold Metrics
2. Ranking Metrics
3. Probability Metrics.

This division is useful because the top metrics used by practitioners for classifiers generally, and specifically imbalanced classification, fit into the taxonomy neatly.

Several machine learning researchers have identified three families of evaluation metrics used in the context of classification. These are the threshold metrics (e.g., accuracy and F-measure), the ranking methods and metrics (e.g., receiver operating characteristics (ROC) analysis and AUC), and the probabilistic metrics (e.g., root-mean-squared error).

— Page 189, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

Let's take a closer look at each group in turn.

4.3.1 Threshold Metrics for Imbalanced Classification

Threshold metrics are those that quantify the classification prediction errors. That is, they are designed to summarize the fraction, ratio, or rate of when a predicted class does not match the expected class in a holdout dataset.

Metrics based on a threshold and a qualitative understanding of error [...] These measures are used when we want a model to minimise the number of errors.

— *An Experimental Comparison Of Performance Measures For Classification*, 2008.

Perhaps the most widely used threshold metric is classification accuracy.

$$\text{Accuracy} = \frac{\text{Correct Predictions}}{\text{Total Predictions}} \quad (4.1)$$

And the complement of classification accuracy called classification error.

$$\text{Error} = \frac{\text{Incorrect Predictions}}{\text{Total Predictions}} \quad (4.2)$$

Although widely used, classification accuracy is almost universally inappropriate for imbalanced classification. The reason is, a high accuracy (or low error) is achievable by a no skill model that only predicts the majority class (for more on the failure of accuracy, see Chapter 5). For imbalanced classification problems, the majority class is typically referred to as the negative outcome (e.g. such as *no change* or *negative test result*), and the minority class is typically referred to as the positive outcome (e.g. *change* or *positive test result*).

- **Majority Class:** Negative outcome, class 0.
- **Minority Class:** Positive outcome, class 1.

Most threshold metrics can be best understood by the terms used in a confusion matrix for a binary (two-class) classification problem. This does not mean that the metrics are limited for use on binary classification; it is just an easy way to quickly understand what is being measured. The confusion matrix provides more insight into not only the performance of a predictive model but also which classes are being predicted correctly, which incorrectly, and what type of errors are being made. In this type of confusion matrix, each cell in the table has a specific and well-understood name, summarized as follows:

	Positive Prediction	Negative Prediction
Positive Class	True Positive (TP)	False Negative (FN)
Negative Class	False Positive (FP)	True Negative (TN)

Listing 4.1: Binary Confusion Matrix.

There are two groups of metrics that may be useful for imbalanced classification because they focus on one class; they are sensitivity-specificity and precision-recall.

Sensitivity-Specificity Metrics

Sensitivity refers to the true positive rate and summarizes how well the positive class was predicted.

$$\text{Sensitivity} = \frac{\text{TruePositive}}{\text{TruePositive} + \text{FalseNegative}} \quad (4.3)$$

Specificity is the complement to sensitivity, or the true negative rate, and summarises how well the negative class was predicted.

$$\text{Specificity} = \frac{\text{TrueNegative}}{\text{FalsePositive} + \text{TrueNegative}} \quad (4.4)$$

For imbalanced classification, the sensitivity might be more interesting than the specificity. Sensitivity and Specificity can be combined into a single score that balances both concerns, called the G-mean.

$$\text{G-mean} = \sqrt{\text{Sensitivity} \times \text{Specificity}} \quad (4.5)$$

Precision-Recall Metrics

Precision summarizes the fraction of examples assigned the positive class that belong to the positive class.

$$\text{Precision} = \frac{\text{TruePositive}}{\text{TruePositive} + \text{FalsePositive}} \quad (4.6)$$

Recall summarizes how well the positive class was predicted and is the same calculation as sensitivity.

$$\text{Recall} = \frac{\text{TruePositive}}{\text{TruePositive} + \text{FalseNegative}} \quad (4.7)$$

Precision and recall can be combined into a single score that seeks to balance both concerns, called the F-score or the F-measure.

$$\text{F-measure} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad (4.8)$$

The F-measure is a popular metric for imbalanced classification. The Fbeta-measure (or $F\beta$ – measure) measure is an abstraction of the F-measure where the balance of precision and recall in the calculation of the harmonic mean is controlled by a coefficient called beta.

$$\text{Fbeta-measure} = \frac{(1 + \beta^2) \times \text{Precision} \times \text{Recall}}{\beta^2 \times \text{Precision} + \text{Recall}} \quad (4.9)$$

For more on precision, recall, and the F-measure, see Chapter 6.

Additional Threshold Metrics

These are probably the most popular metrics to consider, although many others do exist. To give you a taste, these include Kappa, Macro-Average Accuracy, Mean-Class-Weighted Accuracy, Optimized Precision, Adjusted Geometric Mean, Balanced Accuracy, and more. Threshold metrics are easy to calculate and easy to understand. One limitation of these metrics is that they assume that the class distribution observed in the training dataset will match the distribution in the test set and in real data when the model is used to make predictions. This is often the case, but when it is not the case, the performance can be quite misleading.

An important disadvantage of all the threshold metrics discussed in the previous section is that they assume full knowledge of the conditions under which the classifier will be deployed. In particular, they assume that the class imbalance present in the training set is the one that will be encountered throughout the operating life of the classifier

— Page 196, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

Ranking metrics don't make any assumptions about class distributions.

4.3.2 Ranking Metrics for Imbalanced Classification

Rank metrics are more concerned with evaluating classifiers based on how effective they are at separating classes.

Metrics based on how well the model ranks the examples [...] These are important for many applications [...] where classifiers are used to select the best n instances of a set of data or when good class separation is crucial.

— *An Experimental Comparison Of Performance Measures For Classification*, 2008.

These metrics require that a classifier predicts a score or a probability of class membership. From this score, different thresholds can be applied to test the effectiveness of classifiers. Those models that maintain a good score across a range of thresholds will have good class separation and will be ranked higher.

... consider a classifier that gives a numeric score for an instance to be classified in the positive class. Therefore, instead of a simple positive or negative prediction, the score introduces a level of granularity

— Page 53, *Learning from Imbalanced Data Sets*, 2018.

The most commonly used ranking metric is the ROC Curve or ROC Analysis. ROC is an acronym that means Receiver Operating Characteristic and summarizes a field of study for analyzing binary classifiers based on their ability to discriminate classes. A ROC curve is a diagnostic plot for summarizing the behavior of a model by calculating the false positive rate and true positive rate for a set of predictions by the model under different thresholds. The true positive rate is the recall or sensitivity.

$$\text{TruePositiveRate} = \frac{\text{TruePositive}}{\text{TruePositive} + \text{FalseNegative}} \quad (4.10)$$

The false positive rate is calculated as:

$$\text{FalsePositiveRate} = \frac{\text{FalsePositive}}{\text{FalsePositive} + \text{TrueNegative}} \quad (4.11)$$

Each threshold is a point on the plot and the points are connected to form a curve. A classifier that has no skill (e.g. predicts the majority class under all thresholds) will be represented by a diagonal line from the bottom left to the top right. Any points below this line have worse than no skill. A perfect model will be a point in the top left of the plot.

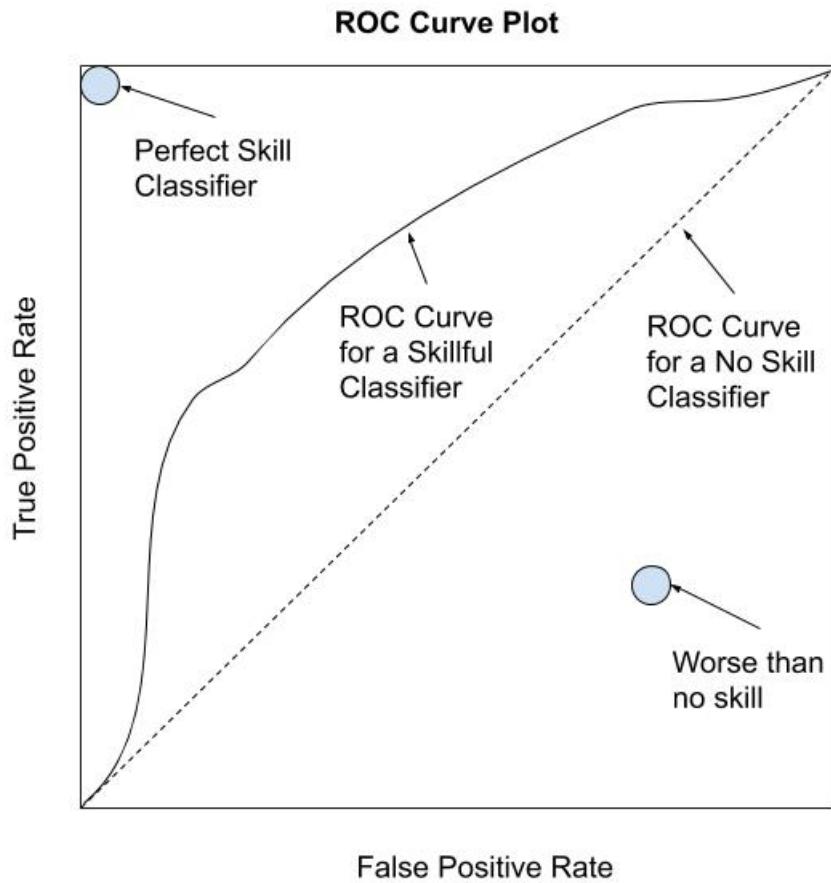


Figure 4.1: Depiction of a ROC Curve.

The ROC Curve is a helpful diagnostic for one model. The area under the ROC curve can be calculated and provides a single score to summarize the plot that can be used to compare models. A no skill classifier will have a score of 0.5, whereas a perfect classifier will have a score of 1.0. For more on ROC Curves and ROC AUC, see Chapter 7.

Although generally effective, the ROC Curve and ROC AUC can be optimistic under a severe class imbalance, especially when the number of examples in the minority class is small. An alternative to the ROC Curve is the precision-recall curve that can be used in a similar way, although focuses on the performance of the classifier on the minority class.

Again, different thresholds are used on a set of predictions by a model, and in this case, the precision and recall are calculated. The points form a curve and classifiers that perform better under a range of different thresholds will be ranked higher. A no-skill classifier will be a horizontal line on the plot with a precision that is proportional to the number of positive examples in the dataset. For a balanced dataset this will be 0.5. A perfect classifier is represented by a point in the top right.

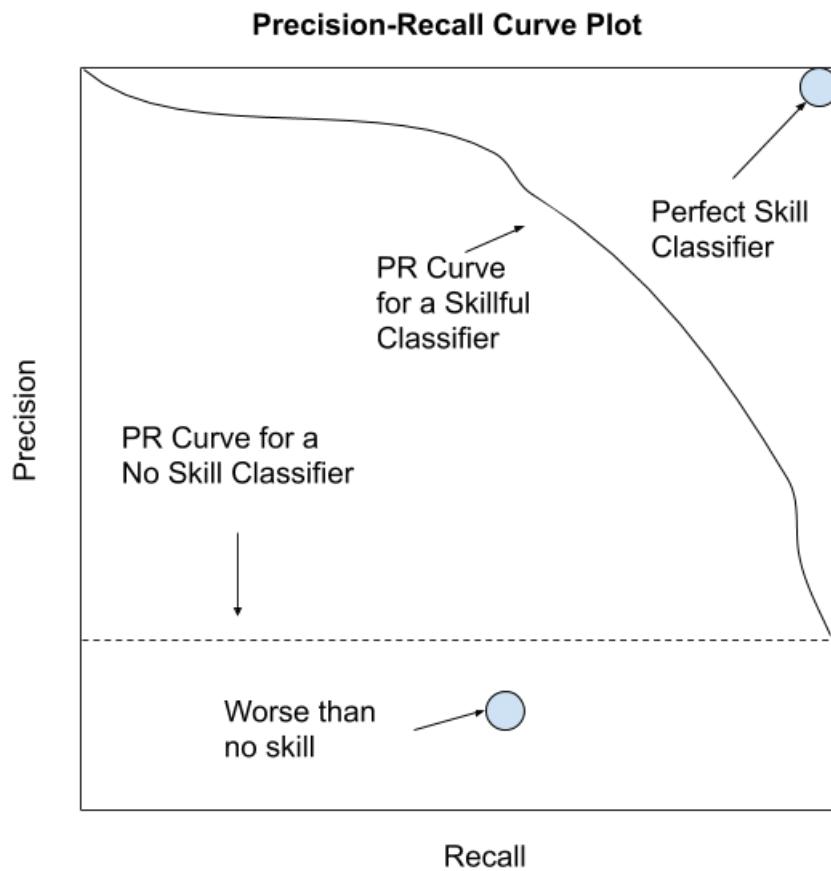


Figure 4.2: Depiction of a Precision-Recall Curve.

Like the ROC Curve, the Precision-Recall Curve is a helpful diagnostic tool for evaluating a single classifier but challenging for comparing classifiers. And like the ROC AUC, we can calculate the area under the curve as a score and use that score to compare classifiers. In this case, the focus on the minority class makes the Precision-Recall AUC more useful for imbalanced classification problems. There are other ranking metrics that are less widely used, such as modification to the ROC Curve for imbalanced classification and cost curves. For more on Precision-Recall Curves and PR AUC, see Chapter 7.

4.3.3 Probabilistic Metrics for Imbalanced Classification

Probabilistic metrics are designed specifically to quantify the uncertainty in a classifier's predictions. These are useful for problems where we are less interested in incorrect vs. correct class predictions and more interested in the uncertainty the model has in predictions and penalizing those predictions that are wrong but highly confident.

Metrics based on a probabilistic understanding of error, i.e. measuring the deviation from the true probability [...] These measures are especially useful when we want an assessment of the reliability of the classifiers, not only measuring when they fail but whether they have selected the wrong class with a high or low probability.

— *An Experimental Comparison Of Performance Measures For Classification*, 2008.

Evaluating a model based on the predicted probabilities requires that the probabilities are calibrated. Some classifiers are trained using a probabilistic framework, such as maximum likelihood estimation, meaning that their probabilities are already calibrated. An example would be logistic regression. Many nonlinear classifiers are not trained under a probabilistic framework and therefore require their probabilities to be calibrated against a dataset prior to being evaluated via a probabilistic metric. Examples might include support vector machines and k -nearest neighbors.

Perhaps the most common metric for evaluating predicted probabilities is log loss for binary classification (or the negative log likelihood), or known more generally as cross-entropy. For a binary classification dataset where the expected values are y and the predicted values are $yhat$, this can be calculated as follows:

$$\text{LogLoss} = -((1 - y) \times \log(1 - yhat) + y \times \log(yhat)) \quad (4.12)$$

The score can be generalized to multiple classes by simply adding the terms; for example:

$$\text{LogLoss} = -\sum_{c \in C} y_c \times \log(yhat_c) \quad (4.13)$$

The score summarizes the average difference between two probability distributions. A perfect classifier has a log loss of 0.0, with worse values being positive up to infinity. For more on log loss, see Chapter 8.

Another popular score for predicted probabilities is the Brier score. The benefit of the Brier score is that it is focused on the positive class, which for imbalanced classification is the minority class. This makes it more preferable than log loss, which is focused on the entire probability distribution. The Brier score is calculated as the mean squared error between the expected probabilities for the positive class (e.g. 1.0) and the predicted probabilities. Recall that the mean squared error is the average of the squared differences between the values.

$$\text{BrierScore} = \frac{1}{N} \times \sum_{i=1}^n (yhat_i - y_i)^2 \quad (4.14)$$

A perfect classifier has a Brier score of 0.0. Although typically described in terms of binary classification tasks, the Brier score can also be calculated for multiclass classification problems. The differences in Brier score for different classifiers can be very small. In order to address this problem, the score can be scaled against a reference score, such as the score from a no skill classifier (e.g. predicting the probability distribution of the positive class in the training dataset). Using the reference score, a Brier Skill Score, or BSS, can be calculated where 0.0 represents no skill, worse than no skill results are negative, and the perfect skill is represented by a value of 1.0.

$$\text{BrierSkillScore} = 1 - \frac{\text{BrierScore}}{\text{BrierScore}_{ref}} \quad (4.15)$$

For more on the Brier Score and Brier Skill Score, see Chapter 8. Although popular for balanced classification problems, probability scoring methods are less widely used for classification problems with a skewed class distribution.

4.4 How to Choose an Evaluation Metric

There is an enormous number of model evaluation metrics to choose from. Given that choosing an evaluation metric is so important and there are tens or perhaps hundreds of metrics to choose from, what are you supposed to do?

The correct evaluation of learned models is one of the most important issues in pattern recognition.

— *An Experimental Comparison Of Performance Measures For Classification*, 2008.

Perhaps the best approach is to talk to project stakeholders and figure out what is important about a model or set of predictions. Then select a few metrics that seem to capture what is important, then test the metric with different scenarios. A scenario might be a mock set of predictions for a test dataset with a skewed class distribution that matches your problem domain. You can test what happens to the metric if a model predicts all the majority class, all the minority class, does well, does poorly, and so on. A few small tests can rapidly help you get a feeling for how the metric might perform.

Another approach might be to perform a literature review and discover what metrics are most commonly used by other practitioners or academics working on the same general type of problem. This can often be insightful, but be warned that some fields of study may fall into groupthink and adopt a metric that might be excellent for comparing large numbers of models at scale, but terrible for model selection in practice.

Still have no idea? Here are some first-order suggestions:

- Are you predicting probabilities?
 - Do you need class labels?
 - * Is the positive class more important?
 - Use Precision-Recall AUC
 - * Are both classes important?
 - Use ROC AUC
 - Do you need probabilities?
 - * Use Brier Score and Brier Skill Score
- Are you predicting class labels?
 - Is the positive class more important?
 - * Are False Negatives and False Positives Equally Important?
 - Use F1-measure
 - * Are False Negatives More Important?
 - Use F2-measure
 - * Are False Positives More Important?
 - Use F0.5-measure
 - Are both classes important?

- * Do you have < 80%-90% Examples for the Majority Class?
 - Use Accuracy
- * Do you have > 80%-90% Examples for the Majority Class?
 - Use G-mean

These suggestions take the important case into account where we might use models that predict probabilities, but require crisp class labels. This is an important class of problems that allow the operator or implementor to choose the threshold to trade-off misclassification errors. In this scenario, error metrics are required that consider all reasonable thresholds, hence the use of the area under curve metrics. We can transform these suggestions into a helpful template.

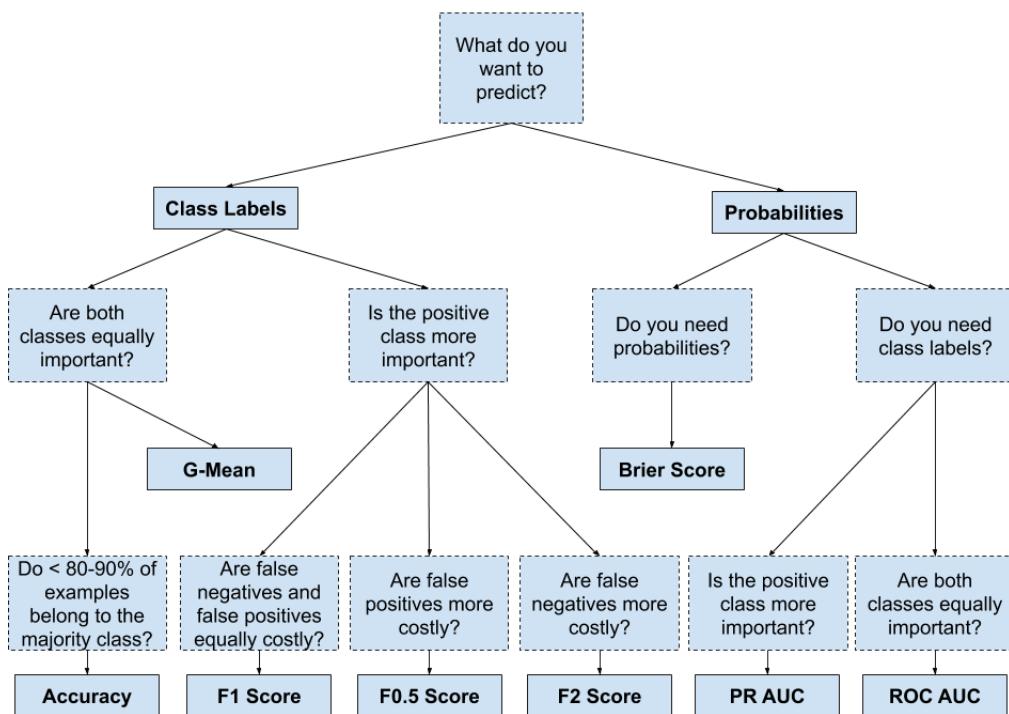


Figure 4.3: How to Choose a Metric for Imbalanced Classification.

4.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

4.5.1 Papers

- *An Experimental Comparison Of Performance Measures For Classification*, 2008.
<https://www.sciencedirect.com/science/article/abs/pii/S0167865508002687>
- *Classification Of Imbalanced Data: A Review*, 2009.
<https://www.worldscientific.com/doi/abs/10.1142/S0218001409007326>
- *A Survey of Predictive Modelling under Imbalanced Distributions*, 2015.
<https://arxiv.org/abs/1505.01658>

4.5.2 Books

- Chapter 8 Assessment Metrics For Imbalanced Learning, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.
<https://amzn.to/32K9K6d>
- Chapter 3 Performance Measures, *Learning from Imbalanced Data Sets*, 2018.
<https://amzn.to/307Xlva>

4.5.3 Articles

- Precision and recall, Wikipedia.
https://en.wikipedia.org/wiki/Precision_and_recall
- Sensitivity and specificity, Wikipedia.
https://en.wikipedia.org/wiki/Sensitivity_and_specificity
- Receiver operating characteristic, Wikipedia.
https://en.wikipedia.org/wiki/Receiver_operating_characteristic
- Cross entropy, Wikipedia.
https://en.wikipedia.org/wiki/Cross_entropy
- Brier score, Wikipedia.
https://en.wikipedia.org/wiki/Brier_score

4.6 Summary

In this tutorial, you discovered metrics that you can use for imbalanced classification. Specifically, you learned:

- About the challenge of choosing metrics for classification, and how it is particularly difficult when there is a skewed class distribution.
- How there are three main types of metrics for evaluating classifier models, referred to as rank, threshold, and probability.
- How to choose a metric for imbalanced classification if you don't know where to start.

4.6.1 Next

In the next tutorial, you will discover the failure of classification accuracy for imbalanced classification.

Chapter 5

The Failure of Accuracy

Classification accuracy is a metric that summarizes the performance of a classification model as the number of correct predictions divided by the total number of predictions. It is easy to calculate and intuitive to understand, making it the most common metric used for evaluating classifier models. This intuition breaks down when the distribution of examples to classes is severely skewed. Intuitions developed by practitioners on balanced datasets, such as 99 percent representing a skillful model, can be incorrect and dangerously misleading on imbalanced classification predictive modeling problems. In this tutorial, you will discover the failure of classification accuracy for imbalanced classification problems. After completing this tutorial, you will know:

- Accuracy and error rate are the de facto standard metrics for summarizing the performance of classification models.
- Classification accuracy fails on classification problems with a skewed class distribution because of the intuitions developed by practitioners on datasets with an equal class distribution.
- Intuition for the failure of accuracy for skewed class distributions with a worked example.

Let's get started.

5.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. What Is Classification Accuracy?
2. Accuracy Fails for Imbalanced Classification
3. Example of Accuracy for Imbalanced Classification

5.2 What Is Classification Accuracy?

Classification predictive modeling involves predicting a class label given examples in a problem domain. The most common metric used to evaluate the performance of a classification predictive model is classification accuracy. Typically, the accuracy of a predictive model is good (above 90% accuracy), therefore it is also very common to summarize the performance of a model in terms of the error rate of the model.

Accuracy and its complement error rate are the most frequently used metrics for estimating the performance of learning systems in classification problems.

— *A Survey of Predictive Modelling under Imbalanced Distributions*, 2015.

Classification accuracy involves first using a classification model to make a prediction for each example in a test dataset. The predictions are then compared to the known labels for those examples in the test set. Accuracy is then calculated as the proportion of examples in the test set that were predicted correctly, divided by all predictions that were made on the test set.

$$\text{Accuracy} = \frac{\text{Correct Predictions}}{\text{Total Predictions}} \quad (5.1)$$

Conversely, the error rate can be calculated as the total number of incorrect predictions made on the test set divided by all predictions made on the test set.

$$\text{Error Rate} = \frac{\text{Incorrect Predictions}}{\text{Total Predictions}} \quad (5.2)$$

The accuracy and error rate are complements of each other, meaning that we can always calculate one from the other. For example:

$$\begin{aligned} \text{Accuracy} &= 1 - \text{Error Rate} \\ \text{Error Rate} &= 1 - \text{Accuracy} \end{aligned} \quad (5.3)$$

Another valuable way to think about accuracy is in terms of the confusion matrix.

A confusion matrix is a summary of the predictions made by a classification model organized into a table by class. Each row of the table indicates the actual class and each column represents the predicted class. Each table cell value is a count of the number of predictions made for a class that are actually for a given class. The cells on the diagonal represent correct predictions, where a predicted and expected class align.

The most straightforward way to evaluate the performance of classifiers is based on the confusion matrix analysis. [...] From such a matrix it is possible to extract a number of widely used metrics for measuring the performance of learning systems, such as Error Rate [...] and Accuracy ...

— *A Study Of The Behavior Of Several Methods For Balancing Machine Learning Training Data*, 2004.

The confusion matrix provides more insight into not only the accuracy of a predictive model, but also which classes are being predicted correctly, which incorrectly, and what type of errors are being made. The simplest confusion matrix is for a two-class classification problem, with negative (class 0) and positive (class 1) classes. In this type of confusion matrix, each cell in the table has a specific and well-understood name, summarized as follows:

	Positive Prediction	Negative Prediction
Positive Class	True Positive (TP)	False Negative (FN)
Negative Class	False Positive (FP)	True Negative (TN)

Listing 5.1: Binary Confusion Matrix.

The classification accuracy can be calculated from this confusion matrix as the sum of correct cells in the table (true positives and true negatives) divided by all cells in the table.

$$\text{Accuracy} = \frac{TP + TN}{TP + FN + FP + TN} \quad (5.4)$$

Similarly, the error rate can also be calculated from the confusion matrix as the sum of incorrect cells of the table (false positives and false negatives) divided by all cells of the table.

$$\text{Error Rate} = \frac{FP + FN}{TP + FN + FP + TN} \quad (5.5)$$

Now that we are familiar with classification accuracy and its complement error rate, let's discover why they might be a bad idea to use for imbalanced classification problems.

5.3 Accuracy Fails for Imbalanced Classification

Classification accuracy is the most-used metric for evaluating classification models. The reason for its wide use is because it is easy to calculate, easy to interpret, and is a single number to summarize the model's capability. As such, it is natural to use it on imbalanced classification problems, where the distribution of examples in the training dataset across the classes is not equal. This is the most common mistake made by beginners to imbalanced classification.

When the class distribution is slightly skewed, accuracy can still be a useful metric. When the skew in the class distributions are severe, accuracy can become an unreliable measure of model performance. The reason for this unreliability is centered around the average machine learning practitioner and the intuitions for classification accuracy. Typically, classification predictive modeling is practiced with small datasets where the class distribution is equal or very close to equal. Therefore, most practitioners develop an intuition that large accuracy score (or conversely small error rate scores) are good, and values above 90 percent are great.

Achieving 90 percent classification accuracy, or even 99 percent classification accuracy, may be trivial on an imbalanced classification problem. This means that intuitions for classification accuracy developed on balanced class distributions will be applied and will be wrong, misleading the practitioner into thinking that a model has good or even excellent performance when it, in fact, does not.

5.3.1 Accuracy Paradox

Consider the case of an imbalanced dataset with a 1:100 class imbalance. In this problem, each example of the minority class (class 1) will have a corresponding 100 examples for the majority class (class 0). In problems of this type, the majority class represents *normal* and the minority class represents *abnormal*, such as a fault, a diagnosis, or a fraud. Good performance on the minority class will be preferred over good performance on both classes.

Considering a user preference bias towards the minority (positive) class examples, accuracy is not suitable because the impact of the least represented, but more important examples, is reduced when compared to that of the majority class.

— *A Survey of Predictive Modelling under Imbalanced Distributions*, 2015.

On this problem, a model that predicts the majority class (class 0) for all examples in the test set will have a classification accuracy of 99 percent, mirroring the distribution of major and minor examples expected in the test set on average. Many machine learning models are designed around the assumption of balanced class distribution, and often learn simple rules (explicit or otherwise) like always predict the majority class, causing them to achieve an accuracy of 99 percent, although in practice performing no better than an unskilled majority class classifier.

A beginner will see the performance of a sophisticated model achieving 99 percent on an imbalanced dataset of this type and believe their work is done, when in fact, they have been misled. This situation is so common that it has a name, referred to as the *accuracy paradox*.

... in the framework of imbalanced data-sets, accuracy is no longer a proper measure, since it does not distinguish between the numbers of correctly classified examples of different classes. Hence, it may lead to erroneous conclusions ...

— *A Review on Ensembles for the Class Imbalance Problem: Bagging-, Boosting-, and Hybrid-Based Approaches*, 2011.

Strictly speaking, accuracy does report a correct result; it is only the practitioner's intuition of high accuracy scores that is the point of failure. Instead of correcting faulty intuitions, it is common to use alternative metrics to summarize model performance for imbalanced classification problems. Now that we are familiar with the idea that classification can be misleading, let's look at a worked example.

5.4 Example of Accuracy for Imbalanced Classification

Although the explanation of why accuracy is a bad idea for imbalanced classification has been given, it is still an abstract idea. We can make the failure of accuracy concrete with a worked example, and attempt to counter any intuitions for accuracy on balanced class distributions that you may have developed, or more likely dissuade the use of accuracy for imbalanced datasets. First, we can define a synthetic dataset with a 1:100 class distribution.

The `make_classification()` scikit-learn function can be used to create a synthetic binary classification dataset. We will specify a `weight` argument of 0.99 with 10,000 examples, this means there will be 9,900 examples for the negative class (class 0) and 100 examples for the positive class (class 1) which is approximately a 1:100 ratio (e.g. $\frac{100}{10000} \times 100$ is 1% and $\frac{9900}{10000} \times 100$ is 99% or 1:99.)

```
...
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
```

Listing 5.2: Example of defining an imbalanced classification dataset.

Next, we can summarize the class distribution to confirm the data has the skew we expected.

```
...
# summarize class distribution
counter = Counter(y)
print(counter)
```

Listing 5.3: Example of summarizing the class distribution.

Finally, we can create a scatter plot of the data points in the data set and color them by class label. This provides a spatial intuition for the skewed class distribution.

```
...
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Listing 5.4: Example of creating a scatter plot of the dataset colored by class label.

Tying this all together, the complete example is listed below.

```
# define an imbalanced dataset with a 1:100 class ratio
from collections import Counter
from sklearn.datasets import make_classification
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
# summarize class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Listing 5.5: Example of creating and summarizing an imbalanced classification dataset.

Running the example first creates the dataset and prints the class distribution. We can see that a little less than 10,000 examples belong to the majority class and 100 examples belong to the minority class, as expected.

```
Counter({0: 9900, 1: 100})
```

Listing 5.6: Example output from creating and summarizing an imbalanced classification dataset.

A plot of the dataset is created and we can see that there are many more examples for the majority class and a helpful legend to indicate the mapping of plot colors to class labels.

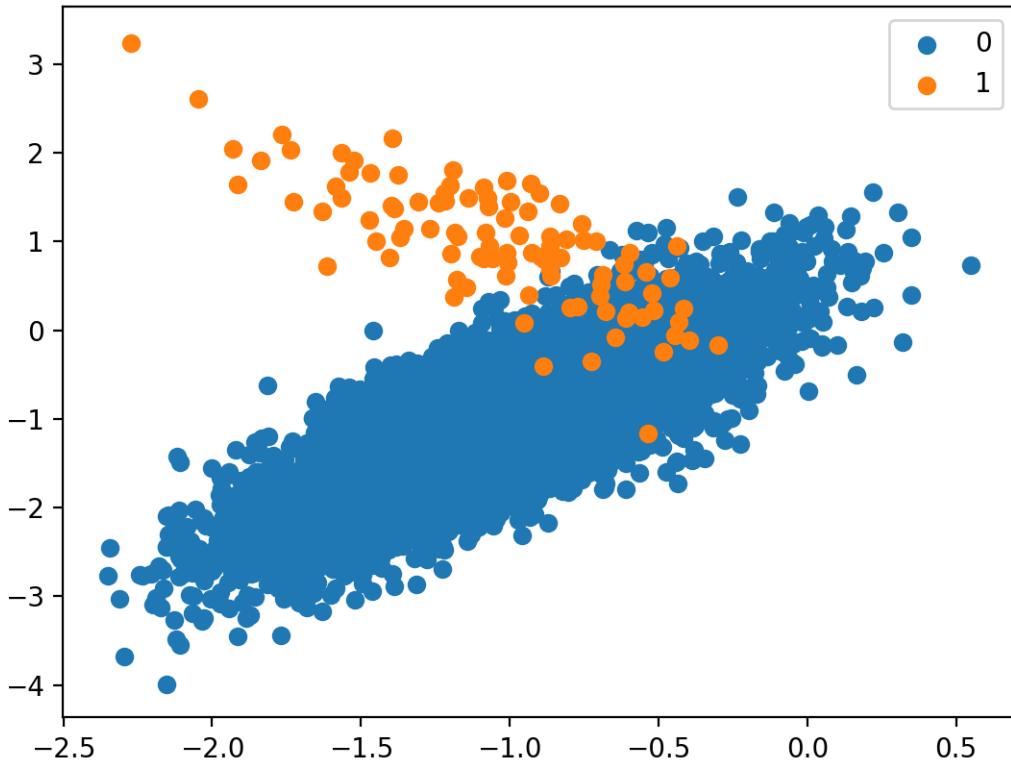


Figure 5.1: Scatter Plot of a Binary Classification Dataset With a 1 to 100 Class Distribution.

Next, we can fit a naive classifier model that always predicts the majority class. We can achieve this using the `DummyClassifier` from scikit-learn and use the ‘`most_frequent`’ strategy that will always predict the class label that is most observed in the training dataset.

```
...
# define model
model = DummyClassifier(strategy='most_frequent')
```

Listing 5.7: Example of defining a naive classification model.

We can then evaluate this model on the training dataset using repeated k -fold cross-validation. It is important that we use stratified cross-validation to ensure that each split of the dataset has the same class distribution as the training dataset. This can be achieved using the `RepeatedStratifiedKFold` class (for more on stratified cross-validation, see Chapter 9). The `evaluate_model()` function below implements this and returns a list of classification accuracy scores for each evaluation of the model.

```
# evaluate a model using repeated k-fold cross-validation
def evaluate_model(X, y, model):
    # define the evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # evaluate the model on the dataset
    scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
```

```
# return scores from each fold and each repeat
return scores
```

Listing 5.8: Example of a function for evaluating a naive classification model.

We can then evaluate the model and calculate the mean of the scores across each evaluation. We would expect that the naive classifier would achieve a classification accuracy of about 99 percent, which we know because that is the distribution of the majority class in the training dataset.

```
...
# evaluate the model
scores = evaluate_model(X, y, model)
# summarize performance
print('Mean Accuracy: %.2f%%' % (mean(scores) * 100))
```

Listing 5.9: Example of reporting mean classification accuracy.

Tying this all together, the complete example of evaluating a naive classifier on the synthetic dataset with a 1:100 class distribution is listed below.

```
# evaluate a majority class classifier on an 1:100 imbalanced dataset
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.dummy import DummyClassifier
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold

# evaluate a model using repeated k-fold cross-validation
def evaluate_model(X, y, model):
    # define the evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # evaluate the model on the dataset
    scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
    # return scores from each fold and each repeat
    return scores

# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
# define model
model = DummyClassifier(strategy='most_frequent')
# evaluate the model
scores = evaluate_model(X, y, model)
# summarize performance
print('Mean Accuracy: %.2f%%' % (mean(scores) * 100))
```

Listing 5.10: Example of evaluating a naive model on an imbalanced classification dataset.

Running the example first reports the class distribution of the training dataset again. Then the model is evaluated and the mean accuracy is reported. We can see that as expected, the performance of the naive classifier matches the class distribution exactly. Normally, achieving 99 percent classification accuracy would be cause for celebration. Although, as we have seen, because the class distribution is imbalanced, 99 percent is actually the lowest acceptable accuracy for this dataset and the starting point from which more sophisticated models must improve.

```
Mean Accuracy: 99.00%
```

Listing 5.11: Example output from evaluating a naive model on an imbalanced classification dataset.

5.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

5.5.1 Papers

- *A Survey of Predictive Modelling under Imbalanced Distributions*, 2015.
<https://arxiv.org/abs/1505.01658>
- *A Review on Ensembles for the Class Imbalance Problem: Bagging-, Boosting-, and Hybrid-Based Approaches*, 2011.
<https://ieeexplore.ieee.org/document/5978225>

5.5.2 Books

- *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.
<https://amzn.to/32K9K6d>
- *Learning from Imbalanced Data Sets*, 2018.
<https://amzn.to/307Xlva>

5.5.3 APIs

- `sklearn.datasets.make_blobs` API.
https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html
- `sklearn.dummy.DummyClassifier` API.
<https://scikit-learn.org/stable/modules/generated/sklearn.dummy.DummyClassifier.html>

5.5.4 Articles

- Accuracy and precision, Wikipedia.
https://en.wikipedia.org/wiki/Accuracy_and_precision
- Accuracy paradox, Wikipedia.
https://en.wikipedia.org/wiki/Accuracy_paradox

5.6 Summary

In this tutorial, you discovered the failure of classification accuracy for imbalanced classification problems. Specifically, you learned:

- Accuracy and error rate are the de facto standard metrics for summarizing the performance of classification models.
- Classification accuracy fails on classification problems with a skewed class distribution because of the intuitions developed by practitioners on datasets with an equal class distribution.
- Intuition for the failure of accuracy for skewed class distributions with a worked example.

5.6.1 Next

In the next tutorial, you will discover precision and recall metrics for evaluating models on imbalanced classification problems.

Chapter 6

Precision, Recall, and F-measure

Classification accuracy is the total number of correct predictions divided by the total number of predictions made for a dataset. As a performance measure, accuracy is inappropriate for imbalanced classification problems. The main reason is that the overwhelming number of examples from the majority class (or classes) will overwhelm the number of examples in the minority class, meaning that even unskillful models can achieve accuracy scores of 90 percent, or 99 percent, depending on how severe the class imbalance happens to be.

An alternative to using classification accuracy is to use precision and recall metrics. In this tutorial, you will discover how to calculate and develop an intuition for precision and recall for imbalanced classification. After completing this tutorial, you will know:

- Precision quantifies the number of positive class predictions that actually belong to the positive class.
- Recall quantifies the number of correct positive class predictions made out of all positive examples in the dataset.
- F-measure provides a single score that balances both the concerns of precision and recall in one number.

Let's get started.

6.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Precision Measure
2. Recall Measure
3. Precision vs. Recall
4. F-measure

6.2 Precision Measure

Precision is a metric that quantifies the number of correct positive predictions made. Precision, therefore, calculates the accuracy for the minority class. It is calculated as the ratio of correctly predicted positive examples divided by the total number of positive examples that were predicted.

Precision evaluates the fraction of correct classified instances among the ones classified as positive ...

— Page 52, *Learning from Imbalanced Data Sets*, 2018.

6.2.1 Precision for Binary Classification

In an imbalanced classification problem with two classes, precision is calculated as the number of true positives divided by the total number of true positives and false positives.

$$\text{Precision} = \frac{\text{TruePositive}}{\text{TruePositive} + \text{FalsePositive}} \quad (6.1)$$

The result is a value between 0.0 for no precision and 1.0 for full or perfect precision. The intuition for precision is that it is not concerned with false negatives and it minimizes false positives. Let's make this calculation concrete with some examples. Consider a dataset with a 1:100 minority to majority ratio, with 100 minority examples and 10,000 majority class examples. A model makes predictions and predicts 120 examples as belonging to the minority class, 90 of which are correct, and 30 of which are incorrect. The precision for this model is calculated as:

$$\begin{aligned} \text{Precision} &= \frac{\text{TruePositive}}{\text{TruePositive} + \text{FalsePositive}} \\ &= \frac{90}{90 + 30} \\ &= \frac{90}{120} \\ &= 0.75 \end{aligned} \quad (6.2)$$

The result is a precision of 0.75, which is a reasonable value but not outstanding. You can see that precision is simply the ratio of correct positive predictions out of all positive predictions made, or the accuracy of minority class predictions. Consider the same dataset, where a model predicts 50 examples belonging to the minority class, 45 of which are true positives and five of which are false positives. We can calculate the precision for this model as follows:

$$\begin{aligned} \text{Precision} &= \frac{\text{TruePositive}}{\text{TruePositive} + \text{FalsePositive}} \\ &= \frac{45}{45 + 5} \\ &= \frac{45}{50} \\ &= 0.90 \end{aligned} \quad (6.3)$$

In this case, although the model predicted far fewer examples as belonging to the minority class, the ratio of correct positive examples is much better. This highlights that although

precision is useful, it does not tell the whole story. It does not comment on how many real positive class examples were predicted as belonging to the negative class, so-called false negatives.

6.2.2 Precision for Multiclass Classification

Precision is not limited to binary classification problems. In an imbalanced classification problem with more than two classes, precision is calculated as the sum of true positives across all classes divided by the sum of true positives and false positives across all classes.

$$\text{Precision} = \frac{\sum_{c \in C} \text{TruePositives}_c}{\sum_{c \in C} \text{TruePositives}_c + \text{FalsePositives}_c} \quad (6.4)$$

For example, we may have an imbalanced multiclass classification problem where the majority class is the negative class, but there are two positive minority classes: class 1 and class 2. Precision can quantify the ratio of correct predictions across both positive classes. Consider a dataset with a 1:1:100 minority to majority class ratio, that is a 1:1 ratio for each positive class and a 1:100 ratio for the minority classes to the majority class, and we have 100 examples in each minority class, and 10,000 examples in the majority class.

A model makes predictions and predicts 70 examples for the first minority class, where 50 are correct and 20 are incorrect. It predicts 150 for the second class with 99 correct and 51 incorrect. Precision can be calculated for this model as follows:

$$\begin{aligned} \text{Precision} &= \frac{\text{TruePositives}_1 + \text{TruePositives}_2}{(\text{TruePositives}_1 + \text{TruePositives}_2) + (\text{FalsePositives}_1 + \text{FalsePositives}_2)} \\ &= \frac{50 + 99}{(50 + 99) + (20 + 51)} \\ &= \frac{149}{149 + 71} \\ &= \frac{149}{220} \\ &= 0.677 \end{aligned} \quad (6.5)$$

We can see that the precision metric calculation scales as we increase the number of minority classes.

6.2.3 Calculate Precision With Scikit-Learn

The precision score can be calculated using the `precision_score()` scikit-learn function. For example, we can use this function to calculate precision for the scenarios in the previous section. First, the case where there are 100 positive to 10,000 negative examples, and a model predicts 90 true positives and 30 false positives. The complete example is listed below.

```
# calculates precision for 1:100 dataset with 90 tp and 30 fp
from sklearn.metrics import precision_score
# define actual
act_pos = [1 for _ in range(100)]
act_neg = [0 for _ in range(10000)]
y_true = act_pos + act_neg
```

```
# define predictions
pred_pos = [0 for _ in range(10)] + [1 for _ in range(90)]
pred_neg = [1 for _ in range(30)] + [0 for _ in range(9970)]
y_pred = pred_pos + pred_neg
# calculate prediction
precision = precision_score(y_true, y_pred, average='binary')
print('Precision: %.3f' % precision)
```

Listing 6.1: Example of calculating precision for a binary classification dataset.

Running the example calculates the precision, matching our manual calculation.

```
Precision: 0.750
```

Listing 6.2: Example output from calculating precision for a binary classification dataset.

Next, we can use the same function to calculate precision for the multiclass problem with 1:1:100, with 100 examples in each minority class and 10,000 in the majority class. A model predicts 50 true positives and 20 false positives for class 1 and 99 true positives and 51 false positives for class 2.

When using the `precision_score()` function for multiclass classification, it is important to specify the minority classes via the `labels` argument and to set the `average` argument to '`micro`' to ensure the calculation is performed as we expect. The complete example is listed below.

```
# calculates precision for 1:1:100 dataset with 50tp,20fp, 99tp,51fp
from sklearn.metrics import precision_score
# define actual
act_pos1 = [1 for _ in range(100)]
act_pos2 = [2 for _ in range(100)]
act_neg = [0 for _ in range(10000)]
y_true = act_pos1 + act_pos2 + act_neg
# define predictions
pred_pos1 = [0 for _ in range(50)] + [1 for _ in range(50)]
pred_pos2 = [0 for _ in range(1)] + [2 for _ in range(99)]
pred_neg = [1 for _ in range(20)] + [2 for _ in range(51)] + [0 for _ in range(9929)]
y_pred = pred_pos1 + pred_pos2 + pred_neg
# calculate prediction
precision = precision_score(y_true, y_pred, labels=[1,2], average='micro')
print('Precision: %.3f' % precision)
```

Listing 6.3: Example of calculating precision for a multiclass classification dataset.

Again, running the example calculates the precision for the multiclass example matching our manual calculation.

```
Precision: 0.677
```

Listing 6.4: Example output from calculating precision for a multiclass classification dataset.

6.3 Recall Measure

Recall is a metric that quantifies the number of correct positive predictions made out of all correct positive predictions that could have been made. Unlike precision that only comments

on the correct positive predictions out of all positive predictions, recall provides an indication of missed positive predictions. In this way, recall provides some notion of the coverage of the positive class.

For imbalanced learning, recall is typically used to measure the coverage of the minority class.

— Page 27, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

6.3.1 Recall for Binary Classification

In an imbalanced classification problem with two classes, recall is calculated as the number of true positives divided by the total number of true positives and false negatives.

$$\text{Recall} = \frac{\text{TruePositive}}{\text{TruePositive} + \text{FalseNegative}} \quad (6.6)$$

The result is a value between 0.0 for no recall and 1.0 for full or perfect recall. The intuition for recall is that it is not concerned with false positives and it minimizes false negatives. Let's make this calculation concrete with some examples. As in the previous section, consider a dataset with 1:100 minority to majority ratio, with 100 minority examples and 10,000 majority class examples. A model makes predictions and predicts 90 of the positive class predictions correctly and 10 incorrectly. We can calculate the recall for this model as follows:

$$\begin{aligned} \text{Recall} &= \frac{\text{TruePositive}}{\text{TruePositive} + \text{FalseNegative}} \\ &= \frac{90}{90 + 10} \\ &= \frac{90}{100} \\ &= 0.9 \end{aligned} \quad (6.7)$$

This model has a good recall.

6.3.2 Recall for Multiclass Classification

Recall is not limited to binary classification problems. In an imbalanced classification problem with more than two classes, recall is calculated as the sum of true positives across all classes divided by the sum of true positives and false negatives across all classes.

$$\text{Recall} = \frac{\sum_{c \in C} \text{TruePositives}_c}{\sum_{c \in C} \text{TruePositives}_c + \text{FalseNegatives}_c} \quad (6.8)$$

As in the previous section, consider a dataset with a 1:1:100 minority to majority class ratio, that is a 1:1 ratio for each positive class and a 1:100 ratio for the minority classes to the majority class, and we have 100 examples in each minority class, and 10,000 examples in the

majority class. A model predicts 77 examples correctly and 23 incorrectly for class 1, and 95 correctly and five incorrectly for class 2. We can calculate recall for this model as follows:

$$\begin{aligned}
 \text{Recall} &= \frac{\text{TruePositives}_1 + \text{TruePositives}_2}{(\text{TruePositives}_1 + \text{TruePositives}_2) + (\text{FalseNegatives}_1 + \text{FalseNegatives}_2)} \\
 &= \frac{77 + 95}{(77 + 95) + (23 + 5)} \\
 &= \frac{172}{172 + 28} \\
 &= \frac{172}{200} \\
 &= 0.86
 \end{aligned} \tag{6.9}$$

6.3.3 Calculate Recall With Scikit-Learn

The recall score can be calculated using the `recall_score()` scikit-learn function. For example, we can use this function to calculate recall for the scenarios above. First, we can consider the case of a 1:100 imbalance with 100 and 10,000 examples respectively, and a model predicts 90 true positives and 10 false negatives. The complete example is listed below.

```
# calculates recall for 1:100 dataset with 90 tp and 10 fn
from sklearn.metrics import recall_score
# define actual
act_pos = [1 for _ in range(100)]
act_neg = [0 for _ in range(10000)]
y_true = act_pos + act_neg
# define predictions
pred_pos = [0 for _ in range(10)] + [1 for _ in range(90)]
pred_neg = [0 for _ in range(10000)]
y_pred = pred_pos + pred_neg
# calculate recall
recall = recall_score(y_true, y_pred, average='binary')
print('Recall: %.3f' % recall)
```

Listing 6.5: Example of calculating recall for a binary classification dataset.

Running the example, we can see that the score matches the manual calculation above.

```
Recall: 0.900
```

Listing 6.6: Example output from calculating recall for a binary classification dataset.

We can also use the `recall_score()` for imbalanced multiclass classification problems. In this case, the dataset has a 1:1:100 imbalance, with 100 in each minority class and 10,000 in the majority class. A model predicts 77 true positives and 23 false negatives for class 1 and 95 true positives and five false negatives for class 2. The complete example is listed below.

```
# calculates recall for 1:1:100 dataset with 77tp,23fn and 95tp,5fn
from sklearn.metrics import recall_score
# define actual
act_pos1 = [1 for _ in range(100)]
act_pos2 = [2 for _ in range(100)]
act_neg = [0 for _ in range(10000)]
```

```

y_true = act_pos1 + act_pos2 + act_neg
# define predictions
pred_pos1 = [0 for _ in range(23)] + [1 for _ in range(77)]
pred_pos2 = [0 for _ in range(5)] + [2 for _ in range(95)]
pred_neg = [0 for _ in range(10000)]
y_pred = pred_pos1 + pred_pos2 + pred_neg
# calculate recall
recall = recall_score(y_true, y_pred, labels=[1,2], average='micro')
print('Recall: %.3f' % recall)

```

Listing 6.7: Example of calculating recall for a multiclass classification dataset.

Again, running the example calculates the recall for the multiclass example matching our manual calculation.

```
Recall: 0.860
```

Listing 6.8: Example output from calculating recall for a multiclass classification dataset.

6.4 Precision vs. Recall

You may decide to use precision or recall on your imbalanced classification problem. Maximizing precision will minimize the number false positive errors, whereas maximizing the recall will minimize the number of false negative errors. As such, precision may be more appropriate on classification problems when false positives are more important. Alternately, recall may be more appropriate on classification problems when false negatives are more important.

- **Precision:** Appropriate when minimizing false positives is the focus.
- **Recall:** Appropriate when minimizing false negatives is the focus.

Sometimes, we want excellent predictions of the positive class. We want high precision and high recall. This can be challenging, as often increases in recall often come at the expense of decreases in precision.

In imbalanced datasets, the goal is to improve recall without hurting precision. These goals, however, are often conflicting, since in order to increase the TP for the minority class, the number of FP is also often increased, resulting in reduced precision.

— Page 55, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

Nevertheless, instead of picking one measure or the other, we can choose a new metric that combines both precision and recall into one score.

6.5 F-measure

Classification accuracy is widely used because it is one single measure used to summarize model performance. F-measure provides a way to combine both precision and recall into a single measure that captures both properties.

Alone, neither precision or recall tells the whole story. We can have excellent precision with terrible recall, or alternately, terrible precision with excellent recall. F-measure provides a way to express both concerns with a single score. Once precision and recall have been calculated for a binary or multiclass classification problem, the two scores can be combined into the calculation of the F-measure. The traditional F-measure is calculated as follows:

$$\text{F-measure} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad (6.10)$$

This is the harmonic mean of the two fractions. This is sometimes called the F-score or the F1-measure and might be the most common metric used on imbalanced classification problems. The intuition for F-measure is that both measures are balanced in importance and that only a good precision and good recall together result in a good F-measure.

... the F1-measure, which weights precision and recall equally, is the variant most often used when learning from imbalanced data.

— Page 27, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

Like precision and recall, a poor F-measure score is 0.0 and a best or perfect F-measure score is 1.0. For example, a perfect precision and recall score would result in a perfect F-measure score:

$$\begin{aligned} \text{F-measure} &= \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \\ &= \frac{2 \times 1.0 \times 1.0}{1.0 + 1.0} \\ &= \frac{2 \times 1.0}{2.0} \\ &= 1.0 \end{aligned} \quad (6.11)$$

Let's make this calculation concrete with a worked example. Consider a binary classification dataset with 1:100 minority to majority ratio, with 100 minority examples and 10,000 majority class examples. Consider a model that predicts 150 examples for the positive class, 95 are correct (true positives), meaning five were missed (false negatives) and 55 are incorrect (false positives). We can calculate the precision as follows:

$$\begin{aligned} \text{Precision} &= \frac{\text{TruePositive}}{\text{TruePositive} + \text{FalsePositive}} \\ &= \frac{95}{95 + 55} \\ &= 0.633 \end{aligned} \quad (6.12)$$

We can calculate the recall as follows:

$$\begin{aligned} \text{Recall} &= \frac{\text{TruePositive}}{\text{TruePositive} + \text{FalseNegative}} \\ &= \frac{95}{95 + 5} \\ &= 0.95 \end{aligned} \tag{6.13}$$

This shows that the model has poor precision, but excellent recall. Finally, we can calculate the F-measure as follows:

$$\begin{aligned} \text{F-measure} &= \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \\ &= \frac{2 \times 0.633 \times 0.95}{0.633 + 0.95} \\ &= \frac{2 \times 0.601}{1.583} \\ &= \frac{1.202}{1.583} \\ &= 0.759 \end{aligned} \tag{6.14}$$

We can see that the good recall levels-out the poor precision, giving an okay or reasonable F-measure score.

6.5.1 Calculate F-measure With Scikit-Learn

The F-measure score can be calculated using the `f1_score()` scikit-learn function. For example, we use this function to calculate F-measure for the scenario above. This is the case of a 1:100 imbalance with 100 and 10,000 examples respectively, and a model predicts 95 true positives, five false negatives, and 55 false positives. The complete example is listed below.

```
# calculates f1 for 1:100 dataset with 95tp, 5fn, 55fp
from sklearn.metrics import f1_score
# define actual
act_pos = [1 for _ in range(100)]
act_neg = [0 for _ in range(10000)]
y_true = act_pos + act_neg
# define predictions
pred_pos = [0 for _ in range(5)] + [1 for _ in range(95)]
pred_neg = [1 for _ in range(55)] + [0 for _ in range(9945)]
y_pred = pred_pos + pred_neg
# calculate score
score = f1_score(y_true, y_pred, average='binary')
print('F-measure: %.3f' % score)
```

Listing 6.9: Example of calculating the F-measure for a binary classification dataset.

Running the example computes the F-measure, matching our manual calculation, within some minor rounding errors.

F-measure: 0.760

Listing 6.10: Example output from calculating the F-measure for a binary classification dataset.

6.5.2 Fbeta-Measure

The F-measure balances the precision and recall. On some problems, we might be interested in an F-measure with more attention put on precision, such as when false positives are more important to minimize, but false negatives are still important. On other problems, we might be interested in an F-measure with more attention put on recall, such as when false negatives are more important to minimize, but false positives are still important.

The solution is the Fbeta-measure ($F\beta$ -measure). The Fbeta-measure is an abstraction of the F-measure where the balance of precision and recall in the calculation of the harmonic mean is controlled by a coefficient called *beta* (β).

$$F\beta = \frac{(1 + \beta^2) \times Precision \times Recall}{\beta^2 \times Precision + Recall} \quad (6.15)$$

The choice of the β parameter will be used in the name of the Fbeta-measure. For example, a β value of 2 is referred to as F2-measure or F2-score. A β value of 1 is referred to as the F1-measure or the F1-score. Three common values for the beta parameter are as follows:

- **F0.5-measure** ($\beta = 0.5$): More weight on precision, less weight on recall.
- **F1-measure** ($\beta = 1$): Balance the weight on precision and recall.
- **F2-measure** ($\beta = 2$): Less weight on precision, more weight on recall.

The scikit-learn library provides the `fbeta_score()` for calculating the Fbeta-measure for a set of predictions and accepts a `beta` argument that can be set to the common values of 0.5 or 2.

6.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

6.6.1 Papers

- *A Systematic Analysis Of Performance Measures For Classification Tasks*, 2009.
<https://www.sciencedirect.com/science/article/abs/pii/S0306457309000259>

6.6.2 Books

- *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.
<https://amzn.to/32K9K6d>
- *Learning from Imbalanced Data Sets*, 2018.
<https://amzn.to/307Xlva>

6.6.3 API

- `sklearn.metrics.precision_score` API.
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_score.html
- `sklearn.metrics.recall_score` API.
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.recall_score.html
- `sklearn.metrics.f1_score` API.
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html
- `sklearn.metrics.fbeta_score` API.
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.fbeta_score.html

6.6.4 Articles

- Confusion matrix, Wikipedia.
https://en.wikipedia.org/wiki/Confusion_matrix
- Precision and recall, Wikipedia.
https://en.wikipedia.org/wiki/Precision_and_recall
- F1 score, Wikipedia.
https://en.wikipedia.org/wiki/F1_score

6.7 Summary

In this tutorial, you discovered how to calculate and develop an intuition for precision and recall for imbalanced classification. Specifically, you learned:

- Precision quantifies the number of positive class predictions that actually belong to the positive class.
- Recall quantifies the number of correct positive class predictions made out of all positive examples in the dataset.
- F-measure provides a single score that balances both the concerns of precision and recall in one number.

6.7.1 Next

In the next tutorial, you will discover ROC curves and ROC AUC performance metrics for evaluating models on imbalanced classification datasets.

Chapter 7

ROC Curves and Precision-Recall Curves

Most imbalanced classification problems involve two classes: a negative case with the majority of examples and a positive case with a minority of examples. Two diagnostic tools that help in the interpretation of binary (two-class) classification predictive models are ROC Curves and Precision-Recall curves.

Plots from the curves can be created and used to understand the trade-off in performance for different threshold values when interpreting probabilistic predictions. Each plot can also be summarized with an area under the curve score that can be used to directly compare classification models. In this tutorial, you will discover ROC Curves and Precision-Recall Curves for imbalanced classification. After completing this tutorial, you will know:

- ROC Curves and Precision-Recall Curves provide a diagnostic tool for binary classification models.
- ROC AUC and Precision-Recall AUC provide scores that summarize the curves and can be used to compare classifiers.
- ROC Curves and ROC AUC can be optimistic on severely imbalanced classification problems with few samples of the minority class.

Let's get started.

7.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. ROC Curves and ROC AUC
2. Precision-Recall Curves and AUC
3. ROC and PR Curves With a Severe Imbalance

7.2 ROC Curves and ROC AUC

An ROC curve (or receiver operating characteristic curve) is a plot that summarizes the performance of a binary classification model on the positive class. The x-axis indicates the False Positive Rate and the y-axis indicates the True Positive Rate.

- **ROC Curve:** Plot of False Positive Rate (x) vs. True Positive Rate (y).

The true positive rate is a fraction calculated as the total number of true positive predictions divided by the sum of the true positives and the false negatives (e.g. all examples in the positive class). The true positive rate is also referred to as the sensitivity or the recall.

$$\text{TruePositiveRate} = \frac{\text{TruePositive}}{\text{TruePositive} + \text{FalseNegative}} \quad (7.1)$$

The false positive rate is calculated as the total number of false positive predictions divided by the sum of the false positives and true negatives (e.g. all examples in the negative class).

$$\text{FalsePositiveRate} = \frac{\text{FalsePositive}}{\text{FalsePositive} + \text{TrueNegative}} \quad (7.2)$$

We can think of the plot as the fraction of correct predictions for the positive class (y-axis) versus the fraction of errors for the negative class (x-axis). Ideally, we want the fraction of correct positive class predictions to be 1 (top of the plot) and the fraction of incorrect negative class predictions to be 0 (left of the plot). This highlights that the best possible classifier that achieves perfect skill is the top-left of the plot (coordinate 0,1).

- **Perfect Skill:** A point in the top left of the plot.

The threshold is applied to the cut-off point in probability between the positive and negative classes, which by default for any classifier would be set at 0.5, halfway between each outcome (0 and 1). A trade-off exists between the TruePositiveRate and FalsePositiveRate, such that changing the threshold of classification will change the balance of predictions towards improving the TruePositiveRate at the expense of FalsePositiveRate, or the reverse case.

By evaluating the true positive and false positives for different threshold values, a curve can be constructed that stretches from the bottom left to top right and bows toward the top left. This curve is called the ROC curve. A classifier that has no discriminative power between positive and negative classes will form a diagonal line between a False Positive Rate of 0 and a True Positive Rate of 0 (coordinate (0,0) or predict all negative class) to a False Positive Rate of 1 and a True Positive Rate of 1 (coordinate (1,1) or predict all positive class). Models represented by points below this line have worse than no skill.

The curve provides a convenient diagnostic tool to investigate one classifier with different threshold values and the effect on the TruePositiveRate and FalsePositiveRate. One might choose a threshold in order to bias the predictive behavior of a classification model. It is a popular diagnostic tool for classifiers on balanced and imbalanced binary prediction problems alike because it is not biased to the majority or minority class.

ROC analysis does not have any bias toward models that perform well on the majority class at the expense of the minority class — a property that is quite attractive when dealing with imbalanced data.

— Page 27, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

We can plot a ROC curve for a model in Python using the `roc_curve()` scikit-learn function. The function takes both the true outcomes (0,1) from the test set and the predicted probabilities for the 1 class. The function returns the false positive rates for each threshold, true positive rates for each threshold and thresholds.

```
...
# calculate roc curve for model
fpr, tpr, _ = roc_curve(testy, pos_probs)
```

Listing 7.1: Example of calculating the ROC curve.

Most scikit-learn models can predict probabilities by calling the `predict_proba()` function. This will return the probabilities for each class, for each sample in a test set, e.g. two numbers for each of the two classes in a binary classification problem. The probabilities for the positive class can be retrieved as the second column in this array of probabilities.

```
...
# predict probabilities
yhat = model.predict_proba(testX)
# retrieve just the probabilities for the positive class
pos_probs = yhat[:, 1]
```

Listing 7.2: Example of predicting probabilities.

We can demonstrate this on a synthetic dataset and plot the ROC curve for a no skill classifier and a Logistic Regression model.

The `make_classification()` function can be used to create synthetic classification problems. In this case, we will create 1,000 examples for a binary classification problem (about 500 examples per class). We will then split the dataset into a train and test sets of equal size in order to fit and evaluate the model.

```
...
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, random_state=1)
# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2)
```

Listing 7.3: Example of creating a dataset and splitting it into train and test sets.

A Logistic Regression model is a good model for demonstration because the predicted probabilities are well-calibrated, as opposed to other machine learning models that are not developed around a probabilistic model, in which case their probabilities may need to be calibrated first (e.g. an SVM).

```
...
# fit a model
model = LogisticRegression(solver='lbfgs')
model.fit(trainX, trainy)
```

Listing 7.4: Example of fitting a logistic regression model

The complete example is listed below.

```

# example of a roc curve for a predictive model
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve
from matplotlib import pyplot
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, random_state=1)
# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2)
# fit a model
model = LogisticRegression(solver='lbfgs')
model.fit(trainX, trainy)
# predict probabilities
yhat = model.predict_proba(testX)
# retrieve just the probabilities for the positive class
pos_probs = yhat[:, 1]
# plot no skill roc curve
pyplot.plot([0, 1], [0, 1], linestyle='--', label='No Skill')
# calculate roc curve for model
fpr, tpr, _ = roc_curve(testy, pos_probs)
# plot model roc curve
pyplot.plot(fpr, tpr, marker='.', label='Logistic')
# axis labels
pyplot.xlabel('False Positive Rate')
pyplot.ylabel('True Positive Rate')
# show the legend
pyplot.legend()
# show the plot
pyplot.show()

```

Listing 7.5: Example of plotting a ROC curve on a balanced dataset.

Running the example creates the synthetic dataset, splits into train and test sets, then fits a Logistic Regression model on the training dataset and uses it to make a prediction on the test set. The ROC Curve for the Logistic Regression model is shown (orange with dots). A no skill classifier as a diagonal line (blue with dashes).

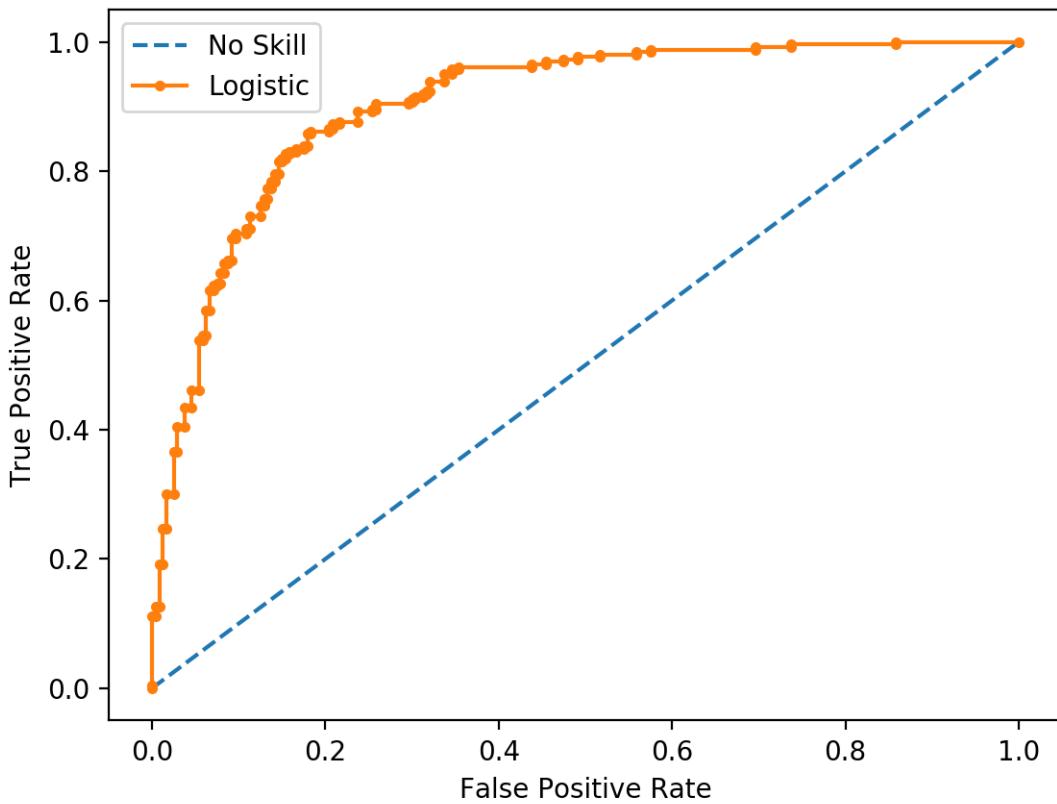


Figure 7.1: ROC Curve of a Logistic Regression Model and a No Skill Classifier.

Now that we have seen the ROC Curve, let's take a closer look at the ROC area under curve score.

7.2.1 ROC Area Under Curve (AUC) Score

Although the ROC Curve is a helpful diagnostic tool, it can be challenging to compare two or more classifiers based on their curves. Instead, the area under the curve can be calculated to give a single score for a classifier model across all threshold values. This is called the ROC area under curve or ROC AUC or sometimes ROCAUC. The score is a value between 0.0 and 1.0, with 1.0 indicating a perfect classifier.

AUCROC can be interpreted as the probability that the scores given by a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative one.

— Page 54, *Learning from Imbalanced Data Sets*, 2018.

This single score can be used to compare binary classifier models directly. As such, this score might be the most commonly used for comparing classification models for imbalanced problems.

The most common metric involves receiver operation characteristics (ROC) analysis, and the area under the ROC curve (AUC).

— Page 27, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

The AUC for the ROC can be calculated in scikit-learn using the `roc_auc_score()` function. Like the `roc_curve()` function, the AUC function takes both the true outcomes (0,1) from the test set and the predicted probabilities for the positive class.

```
...
# calculate roc auc
roc_auc = roc_auc_score(testy, pos_probs)
```

Listing 7.6: Example of calculating the ROC AUC score.

We can demonstrate this the same synthetic dataset with a Logistic Regression model. The complete example is listed below.

```
# example of a roc auc for a predictive model
from sklearn.datasets import make_classification
from sklearn.dummy import DummyClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, random_state=1)
# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2)
# no skill model, stratified random class predictions
model = DummyClassifier(strategy='stratified')
model.fit(trainX, trainy)
yhat = model.predict_proba(testX)
pos_probs = yhat[:, 1]
# calculate roc auc
roc_auc = roc_auc_score(testy, pos_probs)
print('No Skill ROC AUC %.3f' % roc_auc)
# skilled model
model = LogisticRegression(solver='lbfgs')
model.fit(trainX, trainy)
yhat = model.predict_proba(testX)
pos_probs = yhat[:, 1]
# calculate roc auc
roc_auc = roc_auc_score(testy, pos_probs)
print('Logistic ROC AUC %.3f' % roc_auc)
```

Listing 7.7: Example of calculating the ROC AUC on a balanced dataset.

Running the example creates and splits the synthetic dataset, fits the model, and uses the fit model to predict probabilities on the test dataset. In this case, we can see that the ROC AUC for the Logistic Regression model on the synthetic dataset is about 0.903, which is much better than a no skill classifier with a score of about 0.5.

```
No Skill ROC AUC 0.509
Logistic ROC AUC 0.903
```

Listing 7.8: Example output from calculating the ROC AUC on a balanced dataset.

Although widely used, the ROC AUC is not without problems. For imbalanced classification with a severe skew and few examples of the minority class, the ROC AUC can be misleading. This is because a small number of correct or incorrect predictions can result in a large change in the ROC Curve or ROC AUC score.

Although ROC graphs are widely used to evaluate classifiers under presence of class imbalance, it has a drawback: under class rarity, that is, when the problem of class imbalance is associated to the presence of a low sample size of minority instances, as the estimates can be unreliable.

— Page 55, *Learning from Imbalanced Data Sets*, 2018.

A common alternative is the precision-recall curve and area under curve.

7.3 Precision-Recall Curves and AUC

Precision is a metric that quantifies the number of correct positive predictions made. It is calculated as the number of true positives divided by the total number of true positives and false positives.

$$\text{Precision} = \frac{\text{TruePositive}}{\text{TruePositive} + \text{FalsePositive}} \quad (7.3)$$

The result is a value between 0.0 for no precision and 1.0 for full or perfect precision. Recall is a metric that quantifies the number of correct positive predictions made out of all positive predictions. It is calculated as the number of true positives divided by the total number of true positives and false negatives (e.g. it is the true positive rate).

$$\text{Recall} = \frac{\text{TruePositive}}{\text{TruePositive} + \text{FalseNegative}} \quad (7.4)$$

The result is a value between 0.0 for no recall and 1.0 for full or perfect recall. Both the precision and the recall are focused on the positive class (the minority class) and are unconcerned with the true negatives (majority class).

... precision and recall make it possible to assess the performance of a classifier on the minority class.

— Page 27, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

A precision-recall curve (or PR Curve) is a plot of the precision (y-axis) and the recall (x-axis) for different probability thresholds.

- **PR Curve:** Plot of Recall (x) vs Precision (y).

A model with perfect skill is depicted as a point at a coordinate of (1,1). A skillful model is represented by a curve that bows towards a coordinate of (1,1). A no-skill classifier will be a horizontal line on the plot with a precision that is proportional to the number of positive examples in the dataset. For a balanced dataset this will be 0.5. The focus of the PR curve on the minority class makes it an effective diagnostic for imbalanced binary classification models.

Precision-recall curves (PR curves) are recommended for highly skewed domains where ROC curves may provide an excessively optimistic view of the performance.

— *A Survey of Predictive Modelling under Imbalanced Distributions*, 2015.

A precision-recall curve can be calculated in scikit-learn using the `precision_recall_curve()` function that takes the class labels and predicted probabilities for the minority class and returns the precision, recall, and thresholds.

```
...
# calculate precision-recall curve
precision, recall, _ = precision_recall_curve(testy, pos_probs)
```

Listing 7.9: Example of calculating a precision-recall curve.

We can demonstrate this on a synthetic dataset for a predictive model. The complete example is listed below.

```
# example of a precision-recall curve for a predictive model
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_recall_curve
from matplotlib import pyplot
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, random_state=1)
# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2)
# fit a model
model = LogisticRegression(solver='lbfgs')
model.fit(trainX, trainy)
# predict probabilities
yhat = model.predict_proba(testX)
# retrieve just the probabilities for the positive class
pos_probs = yhat[:, 1]
# calculate the no skill line as the proportion of the positive class
no_skill = len(y[y==1]) / len(y)
# plot the no skill precision-recall curve
pyplot.plot([0, 1], [no_skill, no_skill], linestyle='--', label='No Skill')
# calculate model precision-recall curve
precision, recall, _ = precision_recall_curve(testy, pos_probs)
# plot the model precision-recall curve
pyplot.plot(recall, precision, marker='.', label='Logistic')
# axis labels
pyplot.xlabel('Recall')
pyplot.ylabel('Precision')
# show the legend
pyplot.legend()
# show the plot
pyplot.show()
```

Listing 7.10: Example of plotting a precision-recall curve on a balanced dataset.

Running the example creates the synthetic dataset, splits into train and test sets, then fits a Logistic Regression model on the training dataset and uses it to make a prediction on the test

set. The Precision-Recall Curve for the Logistic Regression model is shown (orange with dots). A random or baseline classifier is shown as a horizontal line (blue with dashes).

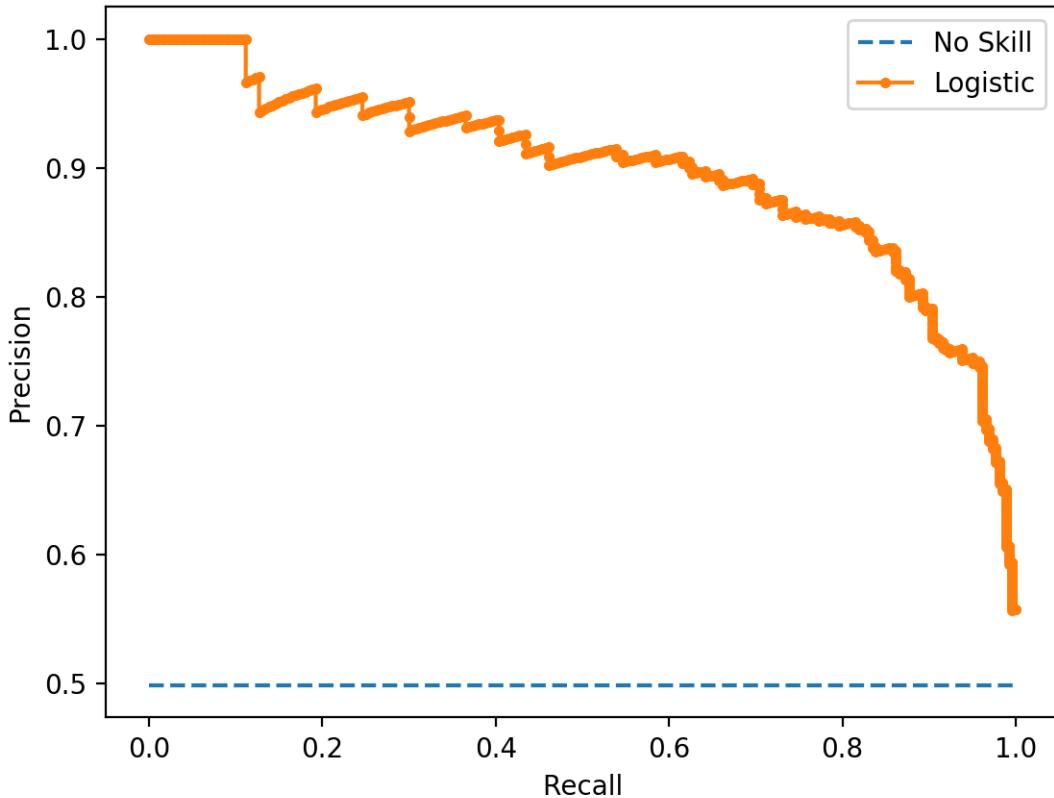


Figure 7.2: Precision-Recall Curve of a Logistic Regression Model and a No Skill Classifier.

Now that we have seen the Precision-Recall Curve, let's take a closer look at the PR area under curve score.

7.3.1 Precision-Recall Area Under Curve (AUC) Score

The Precision-Recall AUC is just like the ROC AUC, in that it summarizes the curve with a range of threshold values as a single score. The score can then be used as a point of comparison between different models on a binary classification problem where a score of 1.0 represents a model with perfect skill. The Precision-Recall AUC score can be calculated using the `auc()` function in scikit-learn, taking the precision and recall values as arguments.

```
...
# calculate the precision-recall auc
auc_score = auc(recall, precision)
```

Listing 7.11: Example of the precision-recall area under curve.

Again, we can demonstrate calculating the Precision-Recall AUC for a Logistic Regression on a synthetic dataset. The complete example is listed below.

```

# example of a precision-recall auc for a predictive model
from sklearn.datasets import make_classification
from sklearn.dummy import DummyClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import auc
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, random_state=1)
# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2)
# no skill model, stratified random class predictions
model = DummyClassifier(strategy='stratified')
model.fit(trainX, trainy)
yhat = model.predict_proba(testX)
pos_probs = yhat[:, 1]
# calculate the precision-recall auc
precision, recall, _ = precision_recall_curve(testy, pos_probs)
auc_score = auc(recall, precision)
print('No Skill PR AUC: %.3f' % auc_score)
# fit a model
model = LogisticRegression(solver='lbfgs')
model.fit(trainX, trainy)
yhat = model.predict_proba(testX)
pos_probs = yhat[:, 1]
# calculate the precision-recall auc
precision, recall, _ = precision_recall_curve(testy, pos_probs)
auc_score = auc(recall, precision)
print('Logistic PR AUC: %.3f' % auc_score)

```

Listing 7.12: Example of calculating the precision-recall area under curve on a balanced dataset.

Running the example creates and splits the synthetic dataset, fits the model, and uses the fit model to predict probabilities on the test dataset. In this case, we can see that the Precision-Recall AUC for the Logistic Regression model on the synthetic dataset is about 0.898, which is much better than a no skill classifier that would achieve the score in this case of 0.632.

No Skill PR AUC: 0.632
Logistic PR AUC: 0.898

Listing 7.13: Example output from calculating the precision-recall area under curve on a balanced dataset.

7.4 ROC and PR Curves With a Severe Imbalance

In this section, we will explore the case of using the ROC Curves and Precision-Recall curves with a binary classification problem that has a severe class imbalance.

Firstly, we can use the `make_classification()` function to create 1,000 examples for a classification problem with about a 1:100 minority to majority class ratio. This can be achieved by setting the `weights` argument and specifying the weighting of generated instances from each class. We will use a 99 percent and 1 percent weighting with 1,000 total examples, meaning there would be about 990 for class 0 and about 10 for class 1.

```
...
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, weights=[0.99, 0.01],
                           random_state=1)
```

Listing 7.14: Example of defining an imbalanced binary classification dataset.

We can then split the dataset into training and test sets and ensure that both have the same general class ratio by setting the `stratify` argument on the call to the `train_test_split()` function and setting it to the array of target variables.

```
...
# split into train/test sets with same class ratio
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2,
                                               stratify=y)
```

Listing 7.15: Example of splitting the dataset into train and test sets.

Tying this together, the complete example of preparing the imbalanced dataset is listed below.

```
# create an imbalanced dataset
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, weights=[0.99, 0.01],
                           random_state=1)
# split into train/test sets with same class ratio
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2,
                                               stratify=y)
# summarize dataset
print('Dataset: Class0=%d, Class1=%d' % (len(y[y==0]), len(y[y==1])))
print('Train: Class0=%d, Class1=%d' % (len(trainy[trainy==0]), len(trainy[trainy==1])))
print('Test: Class0=%d, Class1=%d' % (len(testy[testy==0]), len(testy[testy==1])))
```

Listing 7.16: Example of defining and summarizing the imbalanced dataset.

Running the example first summarizes the class ratio of the whole dataset, then the ratio for each of the train and test sets, confirming the split of the dataset holds the same ratio.

```
Dataset: Class0=985, Class1=15
Train: Class0=492, Class1=8
Test: Class0=493, Class1=7
```

Listing 7.17: Example output from defining and summarizing the imbalanced dataset.

Next, we can develop a Logistic Regression model on the dataset and evaluate the performance of the model using a ROC Curve and ROC AUC score, and compare the results to a no skill classifier, as we did in a prior section. The complete example is listed below.

```
# roc curve and roc auc on an imbalanced dataset
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.dummy import DummyClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score
```

```

from matplotlib import pyplot

# plot no skill and model roc curves
def plot_roc_curve(test_y, naive_probs, model_probs):
    # plot naive skill roc curve
    fpr, tpr, _ = roc_curve(test_y, naive_probs)
    pyplot.plot(fpr, tpr, linestyle='--', label='No Skill')
    # plot model roc curve
    fpr, tpr, _ = roc_curve(test_y, model_probs)
    pyplot.plot(fpr, tpr, marker='.', label='Logistic')
    # axis labels
    pyplot.xlabel('False Positive Rate')
    pyplot.ylabel('True Positive Rate')
    # show the legend
    pyplot.legend()
    # show the plot
    pyplot.show()

# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, weights=[0.99, 0.01],
                           random_state=1)
# split into train/test sets with same class ratio
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2,
                                               stratify=y)
# no skill model, stratified random class predictions
model = DummyClassifier(strategy='stratified')
model.fit(trainX, trainy)
yhat = model.predict_proba(testX)
naive_probs = yhat[:, 1]
# calculate roc auc
roc_auc = roc_auc_score(testy, naive_probs)
print('No Skill ROC AUC %.3f' % roc_auc)
# skilled model
model = LogisticRegression(solver='lbfgs')
model.fit(trainX, trainy)
yhat = model.predict_proba(testX)
model_probs = yhat[:, 1]
# calculate roc auc
roc_auc = roc_auc_score(testy, model_probs)
print('Logistic ROC AUC %.3f' % roc_auc)
# plot roc curves
plot_roc_curve(testy, naive_probs, model_probs)

```

Listing 7.18: Example of ROC curve and AUC for the imbalanced dataset.

Running the example creates the imbalanced binary classification dataset as before. Then a logistic regression model is fit on the training dataset and evaluated on the test dataset. A no skill classifier is evaluated alongside for reference. The ROC AUC scores for both classifiers are reported, showing the no skill classifier achieving the lowest score of approximately 0.5 as expected. The results for the logistic regression model suggest it has some skill with a score of about 0.869.

No Skill ROC AUC 0.490
Logistic ROC AUC 0.869

Listing 7.19: Example output from calculating ROC AUC on the imbalanced dataset.

A ROC curve is also created for the model and the no skill classifier, showing not excellent performance, but definitely skillful performance as compared to the diagonal no skill.

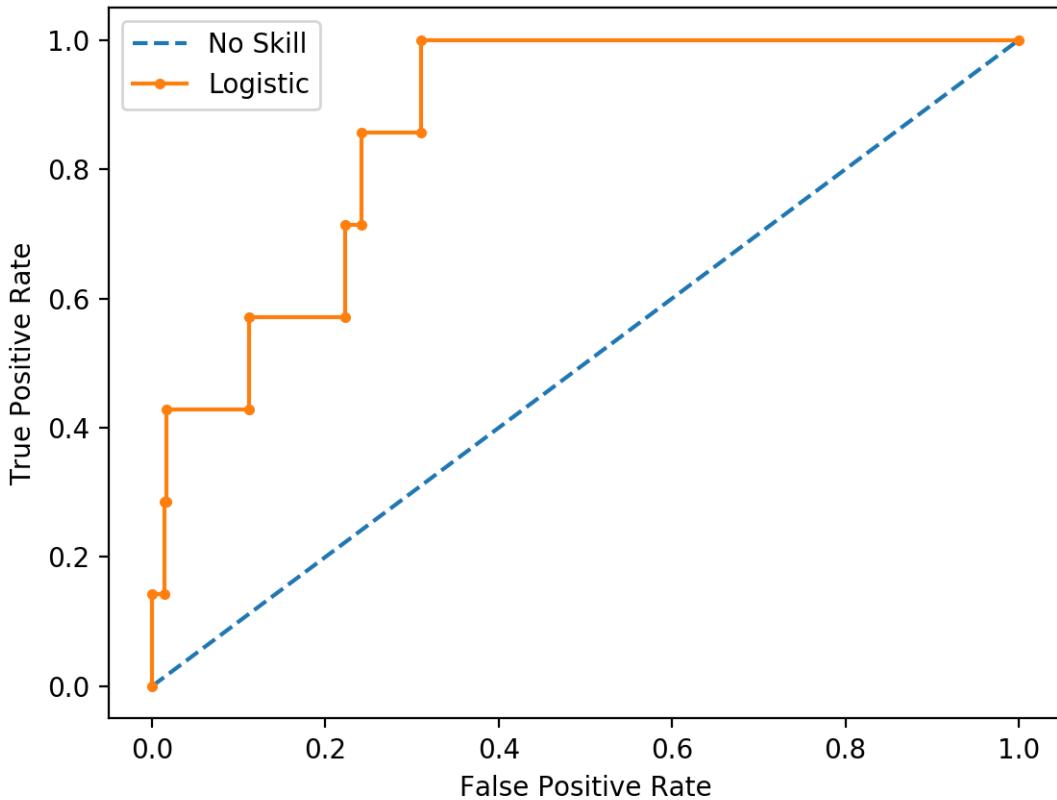


Figure 7.3: Plot of ROC Curve for Logistic Regression on Imbalanced Classification Dataset.

Next, we can perform an analysis of the same model fit and evaluated on the same data using the precision-recall curve and AUC score. The complete example is listed below.

```
# pr curve and pr auc on an imbalanced dataset
from sklearn.datasets import make_classification
from sklearn.dummy import DummyClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import auc
from matplotlib import pyplot

# plot no skill and model precision-recall curves
def plot_pr_curve(test_y, model_probs):
    # calculate the no skill line as the proportion of the positive class
    no_skill = len(test_y[test_y==1]) / len(test_y)
    # plot the no skill precision-recall curve
    pyplot.plot([0, 1], [no_skill, no_skill], linestyle='--', label='No Skill')
    # plot model precision-recall curve
    precision, recall, _ = precision_recall_curve(testy, model_probs)
```

```

pyplot.plot(recall, precision, marker='.', label='Logistic')
# axis labels
pyplot.xlabel('Recall')
pyplot.ylabel('Precision')
# show the legend
pyplot.legend()
# show the plot
pyplot.show()

# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, weights=[0.99, 0.01],
                           random_state=1)
# split into train/test sets with same class ratio
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2,
                                               stratify=y)
# no skill model, stratified random class predictions
model = DummyClassifier(strategy='stratified')
model.fit(trainX, trainy)
yhat = model.predict_proba(testX)
naive_probs = yhat[:, 1]
# calculate the precision-recall auc
precision, recall, _ = precision_recall_curve(testy, naive_probs)
auc_score = auc(recall, precision)
print('No Skill PR AUC: %.3f' % auc_score)
# fit a model
model = LogisticRegression(solver='lbfgs')
model.fit(trainX, trainy)
yhat = model.predict_proba(testX)
model_probs = yhat[:, 1]
# calculate the precision-recall auc
precision, recall, _ = precision_recall_curve(testy, model_probs)
auc_score = auc(recall, precision)
print('Logistic PR AUC: %.3f' % auc_score)
# plot precision-recall curves
plot_pr_curve(testy, model_probs)

```

Listing 7.20: Example of precision-recall curve and AUC for the imbalanced dataset.

As before, running the example creates the imbalanced binary classification dataset. In this case we can see that the Logistic Regression model achieves a PR AUC of about 0.228 and the no skill model achieves a PR AUC of about 0.007.

No Skill PR AUC: 0.007
Logistic PR AUC: 0.228

Listing 7.21: Example output from calculating precision-recall AUC on the imbalanced dataset.

A plot of the precision-recall curve is also created. We can see the horizontal line of the no skill classifier as expected and in this case the zig-zag line of the logistic regression curve close to the no skill line.

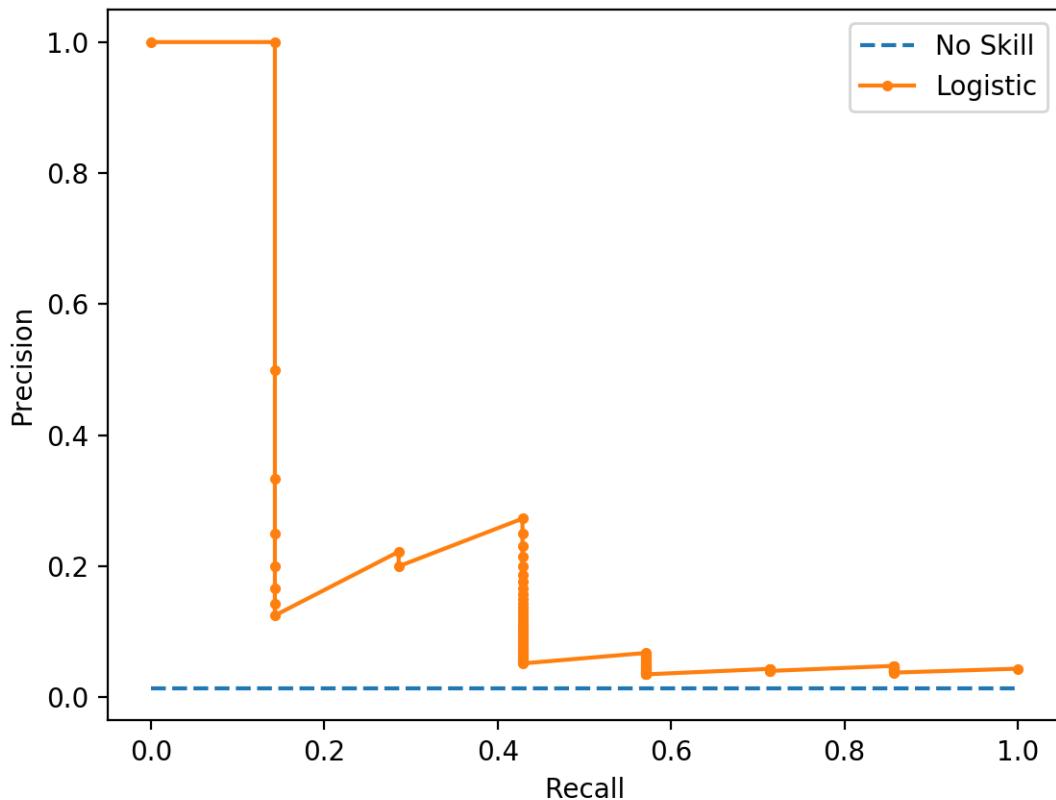


Figure 7.4: Plot of Precision-Recall Curve for Logistic Regression on Imbalanced Classification Dataset.

To explain why the ROC and PR curves tell a different story, recall that the PR curve focuses on the minority class, whereas the ROC curve covers both classes. If we use a threshold of 0.5 and use the logistic regression model to make a prediction for all examples in the test set, we see that it predicts class 0 or the majority class in all cases. This can be confirmed by using the fit model to predict crisp class labels that will use the default threshold of 0.5. The distribution of predicted class labels can then be summarized.

```
...
# predict class labels
yhat = model.predict(testX)
# summarize the distribution of class labels
print(Counter(yhat))
```

Listing 7.22: Example of predicting crisp class labels with the default threshold.

We can then create a histogram of the predicted probabilities of the positive class to confirm that the mass of predicted probabilities is below 0.5, and therefore are mapped to class 0.

```
...
# create a histogram of the predicted probabilities
pyplot.hist(pos_probs, bins=100)
pyplot.show()
```

Listing 7.23: Example of creating a histogram of predicted probabilities for the positive class.

Tying this together, the complete example is listed below.

```
# summarize the distribution of predicted probabilities
from collections import Counter
from matplotlib import pyplot
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, weights=[0.99, 0.01],
                           random_state=1)
# split into train/test sets with same class ratio
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2,
                                               stratify=y)
# fit a model
model = LogisticRegression(solver='lbfgs')
model.fit(trainX, trainy)
# predict probabilities
yhat = model.predict_proba(testX)
# retrieve just the probabilities for the positive class
pos_probs = yhat[:, 1]
# predict class labels
yhat = model.predict(testX)
# summarize the distribution of class labels
print(Counter(yhat))
# create a histogram of the predicted probabilities
pyplot.hist(pos_probs, bins=100)
pyplot.show()
```

Listing 7.24: Example of analyzing the predicted probabilities for the positive class.

Running the example first summarizes the distribution of predicted class labels. As we expected, the majority class (class 0) is predicted for all examples in the test set.

```
Counter({0: 500})
```

Listing 7.25: Example output from summarizing the distribution of predicted probabilities.

A histogram plot of the predicted probabilities for class 1 is also created, showing the center of mass (most predicted probabilities) is less than 0.5 and in fact is generally close to zero.

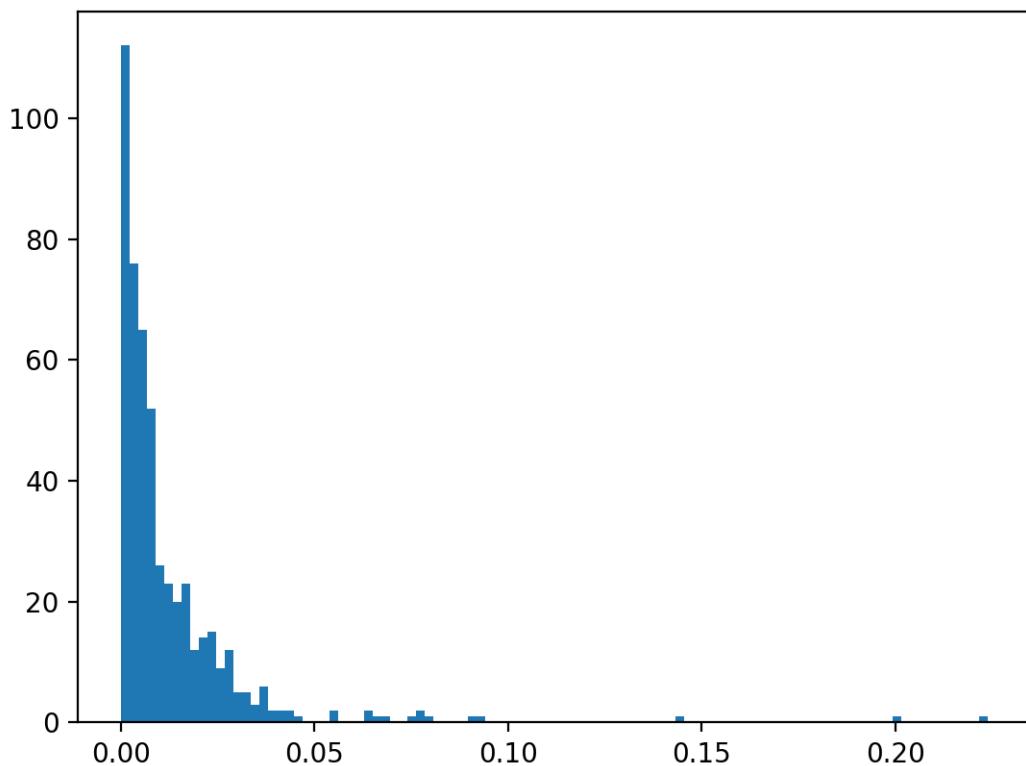


Figure 7.5: Histogram of Logistic Regression Predicted Probabilities for Class 1 for Imbalanced Classification.

This means, unless probability threshold is carefully chosen, any skillful nuance in the predictions made by the model will be lost. Selecting thresholds used to interpret predicted probabilities as crisp class labels is an important topic covered in Chapter 21.

7.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

7.5.1 Papers

- *A Survey of Predictive Modelling under Imbalanced Distributions*, 2015.
<https://arxiv.org/abs/1505.01658>

7.5.2 Books

- *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.
<https://amzn.to/32K9K6d>

- *Learning from Imbalanced Data Sets*, 2018.
<https://amzn.to/307Xlva>

7.5.3 API

- `sklearn.datasets.make_classification` API.
https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification.html
- `sklearn.metrics.roc_curve` API.
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_curve.html
- `sklearn.metrics.roc_auc_score` API.
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_auc_score.html
- `sklearn.metrics.precision_recall_curve` API.
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_curve.html
- `sklearn.metrics.auc` API.
<https://scikit-learn.org/stable/modules/generated/sklearn.metrics.auc.html>

7.5.4 Articles

- Receiver operating characteristic, Wikipedia.
https://en.wikipedia.org/wiki/Receiver_operating_characteristic
- Precision and recall, Wikipedia.
https://en.wikipedia.org/wiki/Precision_and_recall

7.6 Summary

In this tutorial, you discovered ROC Curves and Precision-Recall Curves for imbalanced classification. Specifically, you learned:

- ROC Curves and Precision-Recall Curves provide a diagnostic tool for binary classification models.
- ROC AUC and Precision-Recall AUC provide scores that summarize the curves and can be used to compare classifiers.
- ROC Curves and ROC AUC can be optimistic on severely imbalanced classification problems with few samples of the minority class.

7.6.1 Next

In the next tutorial, you will discover probabilistic performance metrics for evaluating models on imbalanced classification datasets.

Chapter 8

Probability Scoring Methods

Classification predictive modeling involves predicting a class label for examples, although some problems require the prediction of a probability of class membership. For these problems, the crisp class labels are not required, and instead, the likelihood that each example belonging to each class is required and later interpreted. As such, small relative probabilities can carry a lot of meaning and specialized metrics are required to quantify the predicted probabilities. In this tutorial, you will discover metrics for evaluating probabilistic predictions for imbalanced classification. After completing this tutorial, you will know:

- Probability predictions are required for some classification predictive modeling problems.
- Log loss quantifies the average difference between predicted and expected probability distributions.
- Brier score quantifies the average difference between predicted and expected probabilities.

Let's get started.

8.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Probability Metrics
2. Log Loss Score
3. Brier Score

8.2 Probability Metrics

Classification predictive modeling involves predicting a class label for an example. On some problems, a crisp class label is not required, and instead a probability of class membership is preferred. The probability summarizes the likelihood (or uncertainty) of an example belonging to each class label. Probabilities are more nuanced and can be interpreted by a human operator or a system in decision making.

Probability metrics are those specifically designed to quantify the skill of a classifier model using the predicted probabilities instead of crisp class labels. They are typically scores that provide a single value that can be used to compare different models based on how well the predicted probabilities match the expected class probabilities. In practice, a dataset will not have target probabilities. Instead, it will have class labels.

For example, a two-class (binary) classification problem will have the class labels 0 for the negative case and 1 for the positive case. When an example has the class label 0, then the probability of the class labels 0 and 1 will be 1 and 0 respectively. When an example has the class label 1, then the probability of class labels 0 and 1 will be 0 and 1 respectively.

- **Example with Class = 0:** $P(class = 0) = 1, P(class = 1) = 0$.
- **Example with Class = 1:** $P(class = 0) = 0, P(class = 1) = 1$.

We can see how this would scale to three classes or more; for example:

- **Example with Class = 0:** $P(class = 0) = 1, P(class = 1) = 0, P(class = 2) = 0$.
- **Example with Class = 1:** $P(class = 0) = 0, P(class = 1) = 1, P(class = 2) = 0$.
- **Example with Class = 2:** $P(class = 0) = 0, P(class = 1) = 0, P(class = 2) = 1$.

In the case of binary classification problems, this representation can be simplified to just focus on the positive class. That is, we only require the probability of an example belonging to class 1 to represent the probabilities for binary classification (e.g. the Bernoulli distribution); for example:

- **Example with Class = 0:** $P(class = 1) = 0$
- **Example with Class = 1:** $P(class = 1) = 1$

Probability metrics will summarize how well the predicted distribution of class membership matches the known class probability distribution. This focus on predicted probabilities may mean that the crisp class labels predicted by a model are ignored. This focus may mean that a model that predicts probabilities may appear to have terrible performance when evaluated according to its crisp class labels, such as using accuracy or a similar score. This is because although the predicted probabilities may show skill, they must be interpreted with a threshold prior to being converted into crisp class labels.

Additionally, the focus on predicted probabilities may also require that the probabilities predicted by some nonlinear models to be calibrated prior to being used or evaluated. Some models will learn calibrated probabilities as part of the training process (e.g. logistic regression), but many will not and will require calibration (e.g. support vector machines, decision trees, and neural networks). We will cover the topic of calibrating predicted probabilities in Chapter 22. A given probability metric is typically calculated for each example, then averaged across all examples in the training dataset. There are two popular metrics for evaluating predicted probabilities; they are:

- Log Loss
- Brier Score

Let's take a closer look at each in turn.

8.3 Log Loss Score

Logarithmic loss or log loss for short is a loss function known for training the logistic regression classification algorithm. The log loss function calculates the negative log likelihood for probability predictions made by the binary classification model. Most notably, this is logistic regression, but this function can be used by other models, such as neural networks, and is known by other names, such as cross-entropy. Generally, the log loss can be calculated using the expected probabilities P for each class and the natural logarithm of the predicted probabilities Q for each class; for example:

$$\text{LogLoss} = -(P(\text{class} = 0) \times \log(Q(\text{class} = 0)) + (P(\text{class} = 1)) \times \log(Q(\text{class} = 1))) \quad (8.1)$$

The best possible log loss is 0.0, and values are positive to infinite for progressively worse scores. If you are just predicting the probability for the positive class, then the log loss function can be calculated for one binary classification prediction ($yhat$) compared to the expected probability (y) as follows:

$$\text{LogLoss} = -((1 - y) \times \log(1 - yhat) + y \times \log(yhat)) \quad (8.2)$$

For example, if the expected probability was 1.0 and the model predicted 0.8, the log loss would be:

$$\begin{aligned} \text{LogLoss} &= -((1 - y) \times \log(1 - yhat) + y \times \log(yhat)) \\ &= -((1 - 1.0) \times \log(1 - 0.8) + 1.0 \times \log(0.8)) \\ &= -(0.0 + -0.223) \\ &= 0.223 \end{aligned} \quad (8.3)$$

This calculation can be scaled up for multiple classes by adding additional terms; for example:

$$\text{LogLoss} = -\left(\sum_{c \in C} y_c \times \log(yhat_c)\right) \quad (8.4)$$

This generalization is also known as cross-entropy and calculates the number of bits (if log base-2 is used) or nats (if log base-e is used) by which two probability distributions differ. Specifically, it builds upon the idea of entropy from information theory and calculates the average number of bits required to represent or transmit an event from one distribution compared to the other distribution.

... the cross entropy is the average number of bits needed to encode data coming from a source with distribution p when we use model q ...

— Page 57, *Machine Learning: A Probabilistic Perspective*, 2012.

The intuition for this definition comes if we consider a target or underlying probability distribution P and an approximation of the target distribution Q , then the cross-entropy of Q from P is the number of additional bits to represent an event using Q instead of P . We will stick with log loss for now, as it is the term most commonly used when using this calculation as an evaluation metric for classifier models. When calculating the log loss for a set of predictions

compared to a set of expected probabilities in a test dataset, the average of the log loss across all samples is calculated and reported; for example:

$$\text{AverageLogLoss} = \frac{1}{N} \times \sum_{i=1}^N -((1 - y_i) \times \log(1 - yhat_i) + y_i \times \log(yhat_i)) \quad (8.5)$$

The average log loss for a set of predictions on a training dataset is often simply referred to as the log loss. We can demonstrate calculating log loss with a worked example. First, let's define a synthetic binary classification dataset. We will use the `make_classification()` function to create 1,000 examples, with 99%/1% split for the two classes. The complete example of creating and summarizing the dataset is listed below.

```
# create an imbalanced dataset
from numpy import unique
from sklearn.datasets import make_classification
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, weights=[0.99], flip_y=0,
    random_state=1)
# summarize dataset
classes = unique(y)
total = len(y)
for c in classes:
    n_examples = len(y[y==c])
    percent = n_examples / total * 100
    print('> Class=%d : %d/%d (%.1f%%)' % (c, n_examples, total, percent))
```

Listing 8.1: Example of defining and summarizing the imbalanced dataset.

Running the example creates the dataset and reports the distribution of examples in each class.

```
> Class=0 : 990/1000 (99.0%)
> Class=1 : 10/1000 (1.0%)
```

Listing 8.2: Example output from summarizing the imbalanced dataset.

Next, we will develop an intuition for naive predictions of probabilities. A naive prediction strategy would be to predict certainty for the majority class, or $P(class = 0) = 1$. An alternative strategy would be to predict the minority class, or $P(class = 1) = 1$. Log loss can be calculated using the `log_loss()` scikit-learn function. It takes the predicted probability for each class as input and returns the average log loss. Specifically, each example must have a prediction with one probability per class, meaning a prediction for one example for a binary classification problem must have a probability for class 0 and class 1. Therefore, predicting certain probabilities for class 0 for all examples would be implemented as follows:

```
...
# no skill prediction 0
probabilities = [[1, 0] for _ in range(len(testy))]
avg_logloss = log_loss(testy, probabilities)
print('P(class0=1): Log Loss=%.3f' % (avg_logloss))
```

Listing 8.3: Example of predicting certain probabilities.

We can do the same thing for $P(class1) = 1$. These two strategies are expected to perform terribly. A better naive strategy would be to predict the class distribution for each example. For

example, because our dataset has a 99%/1% class distribution for the majority and minority classes, this distribution can be *predicted* for each example to give a baseline for probability predictions.

```
...
# baseline probabilities
probabilities = [[0.99, 0.01] for _ in range(len(testy))]
avg_logloss = log_loss(testy, probabilities)
print('Baseline: Log Loss=% .3f' % (avg_logloss))
```

Listing 8.4: Example of predicting naive probabilities.

Finally, we can also calculate the log loss for perfectly predicted probabilities by taking the target values for the test set as predictions.

```
...
# perfect probabilities
avg_logloss = log_loss(testy, testy)
print('Perfect: Log Loss=% .3f' % (avg_logloss))
```

Listing 8.5: Example of calculating log loss.

Tying this all together, the complete example is listed below.

```
# log loss for naive probability predictions.
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import log_loss
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, weights=[0.99], flip_y=0,
    random_state=1)
# split into train/test sets with same class ratio
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2,
    stratify=y)
# no skill prediction 0
probabilities = [[1, 0] for _ in range(len(testy))]
avg_logloss = log_loss(testy, probabilities)
print('P(class0=1): Log Loss=% .3f' % (avg_logloss))
# no skill prediction 1
probabilities = [[0, 1] for _ in range(len(testy))]
avg_logloss = log_loss(testy, probabilities)
print('P(class1=1): Log Loss=% .3f' % (avg_logloss))
# baseline probabilities
probabilities = [[0.99, 0.01] for _ in range(len(testy))]
avg_logloss = log_loss(testy, probabilities)
print('Baseline: Log Loss=% .3f' % (avg_logloss))
# perfect probabilities
avg_logloss = log_loss(testy, testy)
print('Perfect: Log Loss=% .3f' % (avg_logloss))
```

Listing 8.6: Example of calculating log loss for different naive prediction models.

Running the example reports the log loss for each naive strategy. As expected, predicting certainty for each class label is punished with large log loss scores, with the case of being certain for the minority class in all cases resulting in a much larger score. We can see that predicting the distribution of examples in the dataset as the baseline results in a better score than either of the other naive measures. This baseline represents the no skill classifier and log loss scores

below this strategy represent a model that has some skill. Finally, we can see that a log loss for perfectly predicted probabilities is 0.0, indicating no difference between actual and predicted probability distributions.

```
P(class0=1): Log Loss=0.345
P(class1=1): Log Loss=34.193
Baseline: Log Loss=0.056
Perfect: Log Loss=0.000
```

Listing 8.7: Example output from calculating log loss for different naive prediction models.

Now that we are familiar with log loss, let's take a look at the Brier score.

8.4 Brier Score

The Brier score, named for Glenn Brier, calculates the mean squared error between predicted probabilities and the expected values. The score summarizes the magnitude of the error in the predicted probabilities and is designed for binary classification problems. It is focused on evaluating the probabilities for the positive class. Nevertheless, it can be adapted for problems with multiple classes. It is also an appropriate probabilistic metric for imbalanced classification problems.

The evaluation of probabilistic scores is generally performed by means of the Brier Score. The basic idea is to compute the mean squared error (MSE) between predicted probability scores and the true class indicator, where the positive class is coded as 1, and negative class 0.

— Page 57, *Learning from Imbalanced Data Sets*, 2018.

The error score is always between 0.0 and 1.0, where a model with perfect skill has a score of 0.0. The Brier score can be calculated for positive predicted probabilities (y_{hat}) compared to the expected probabilities (y) as follows:

$$\text{BrierScore} = \frac{1}{N} \times \sum_{i=1}^N (y_{\text{hat}_i} - y_i)^2 \quad (8.6)$$

For example, if a predicted positive class probability is 0.8 and the expected probability is 1.0, then the Brier score is calculated as:

$$\begin{aligned} \text{BrierScore} &= (y_{\text{hat}_i} - y_i)^2 \\ &= (0.8 - 1.0)^2 \\ &= 0.04 \end{aligned} \quad (8.7)$$

We can demonstrate calculating the Brier score with a worked example using the same dataset and naive predictive models as were used in the previous section. The Brier score can be calculated using the `brier_score_loss()` scikit-learn function. It takes the probabilities for the positive class only, and returns an average score. As in the previous section, we can evaluate naive strategies of predicting the certainty for each class label. In this case, as the score only considers the probability for the positive class, this will involve predicting 0.0 for $P(\text{class} = 1) = 0$ and 1.0 for $P(\text{class} = 1) = 1$. For example:

```
...
# no skill prediction 0
probabilities = [0.0 for _ in range(len(testy))]
avg_brier = brier_score_loss(testy, probabilities)
print('P(class1=0): Brier Score=% .4f' % (avg_brier))
# no skill prediction 1
probabilities = [1.0 for _ in range(len(testy))]
avg_brier = brier_score_loss(testy, probabilities)
print('P(class1=1): Brier Score=% .4f' % (avg_brier))
```

Listing 8.8: Example of naive models for Brier score.

We can also test the no skill classifier that predicts the ratio of positive examples in the dataset, which in this case is 1 percent or 0.01.

```
...
# baseline probabilities
probabilities = [0.01 for _ in range(len(testy))]
avg_brier = brier_score_loss(testy, probabilities)
print('Baseline: Brier Score=% .4f' % (avg_brier))
```

Listing 8.9: Example of baseline model for Brier score.

Finally, we can also confirm the Brier score for perfectly predicted probabilities.

```
...
# perfect probabilities
avg_brier = brier_score_loss(testy, testy)
print('Perfect: Brier Score=% .4f' % (avg_brier))
```

Listing 8.10: Example of perfect predictions for Brier score.

Tying this together, the complete example is listed below.

```
# brier score for naive probability predictions.
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import brier_score_loss
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, weights=[0.99], flip_y=0,
                           random_state=1)
# split into train/test sets with same class ratio
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2,
                                               stratify=y)
# no skill prediction 0
probabilities = [0.0 for _ in range(len(testy))]
avg_brier = brier_score_loss(testy, probabilities)
print('P(class1=0): Brier Score=% .4f' % (avg_brier))
# no skill prediction 1
probabilities = [1.0 for _ in range(len(testy))]
avg_brier = brier_score_loss(testy, probabilities)
print('P(class1=1): Brier Score=% .4f' % (avg_brier))
# baseline probabilities
probabilities = [0.01 for _ in range(len(testy))]
avg_brier = brier_score_loss(testy, probabilities)
print('Baseline: Brier Score=% .4f' % (avg_brier))
# perfect probabilities
```

```
avg_brier = brier_score_loss(testy, testy)
print('Perfect: Brier Score=% .4f' % (avg_brier))
```

Listing 8.11: Example of calculating Brier score for different naive prediction models.

Running the example, we can see the scores for the naive models and the baseline no skill classifier. As we might expect, we can see that predicting a 0.0 for all examples results in a low score, as the mean squared error between all 0.0 predictions and mostly 0 classes in the test set results in a small value. Conversely, the error between 1.0 predictions and mostly 0 class values results in a larger error score. Importantly, we can see that the default no skill classifier results in a lower score than predicting all 0.0 values. Again, this represents the baseline score, below which models will demonstrate skill.

```
P(class1=0): Brier Score=0.0100
P(class1=1): Brier Score=0.9900
Baseline: Brier Score=0.0099
Perfect: Brier Score=0.0000
```

Listing 8.12: Example output from calculating Brier score for different naive prediction models.

The Brier scores can become very small and the focus will be on fractions well below the decimal point. For example, the difference in the above example between Baseline and Perfect scores is slight at four decimal places. A common practice is to transform the score using a reference score, such as the no skill classifier. This is called a Brier Skill Score, or BSS, and is calculated as follows:

$$\text{BrierSkillScore} = 1 - \frac{\text{BrierScore}}{\text{BrierScore}_{ref}} \quad (8.8)$$

We can see that if the reference score was evaluated, it would result in a BSS of 0.0. This represents a no skill prediction. Values below this will be negative and represent worse than no skill. Values above 0.0 represent skillful predictions with a perfect prediction value of 1.0. We can demonstrate this by developing a function to calculate the Brier skill score listed below.

```
# calculate the brier skill score
def brier_skill_score(y, yhat, brier_ref):
    # calculate the brier score
    bs = brier_score_loss(y, yhat)
    # calculate skill score
    return 1.0 - (bs / brier_ref)
```

Listing 8.13: Example function for calculating Brier Skill Score.

We can then calculate the BSS for each of the naive predictions, as well as for a perfect prediction. The complete example is listed below.

```
# brier skill score for naive probability predictions.
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import brier_score_loss

# calculate the brier skill score
def brier_skill_score(y, yhat, brier_ref):
    # calculate the brier score
    bs = brier_score_loss(y, yhat)
```

```

# calculate skill score
return 1.0 - (bs / brier_ref)

# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, weights=[0.99], flip_y=0,
                           random_state=1)
# split into train/test sets with same class ratio
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2,
                                               stratify=y)
# calculate reference
probabilities = [0.01 for _ in range(len(testy))]
brier_ref = brier_score_loss(testy, probabilities)
print('Reference: Brier Score=% .4f' % (brier_ref))
# no skill prediction 0
probabilities = [0.0 for _ in range(len(testy))]
bss = brier_skill_score(testy, probabilities, brier_ref)
print('P(class1=0): BSS=% .4f' % (bss))
# no skill prediction 1
probabilities = [1.0 for _ in range(len(testy))]
bss = brier_skill_score(testy, probabilities, brier_ref)
print('P(class1=1): BSS=% .4f' % (bss))
# baseline probabilities
probabilities = [0.01 for _ in range(len(testy))]
bss = brier_skill_score(testy, probabilities, brier_ref)
print('Baseline: BSS=% .4f' % (bss))
# perfect probabilities
bss = brier_skill_score(testy, testy, brier_ref)
print('Perfect: BSS=% .4f' % (bss))

```

Listing 8.14: Example of calculating Brier Skill Score for different naive prediction models.

Running the example first calculates the reference Brier score used in the BSS calculation. We can then see that predicting certainty scores for each class results in a negative BSS score, indicating that they are worse than no skill. Finally, we can see that evaluating the reference forecast itself results in 0.0, indicating no skill and evaluating the true values as predictions results in a perfect score of 1.0. As such, the Brier Skill Score is a best practice for evaluating probability predictions and is widely used where probability classification prediction are evaluated routinely, such as in weather forecasts (e.g. rain or not).

```

Reference: Brier Score=0.0099
P(class1=0): BSS=-0.0101
P(class1=1): BSS=-99.0000
Baseline: BSS=0.0000
Perfect: BSS=1.0000

```

Listing 8.15: Example output from calculating Brier Skill Score for different naive prediction models.

8.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

8.5.1 Books

- Chapter 8 Assessment Metrics For Imbalanced Learning, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.
<https://amzn.to/32K9K6d>
- Chapter 3 Performance Measures, *Learning from Imbalanced Data Sets*, 2018.
<https://amzn.to/307Xlva>

8.5.2 API

- `sklearn.datasets.make_classification` API.
https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification.html
- `sklearn.metrics.log_loss` API.
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.log_loss.html
- `sklearn.metrics.brier_score_loss` API.
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.brier_score_loss.html

8.5.3 Articles

- Brier score, Wikipedia.
https://en.wikipedia.org/wiki/Brier_score
- Cross entropy, Wikipedia.
https://en.wikipedia.org/wiki/Cross_entropy
- Joint Working Group on Forecast Verification Research.
<https://www.cawcr.gov.au/projects/verification/>

8.6 Summary

In this tutorial, you discovered metrics for evaluating probabilistic predictions for imbalanced classification. Specifically, you learned:

- Probability predictions are required for some classification predictive modeling problems.
- Log loss quantifies the average difference between predicted and expected probability distributions.
- Brier score quantifies the average difference between predicted and expected probabilities.

8.6.1 Next

In the next tutorial, you will discover how to correctly use cross-validation and train-test sets on imbalanced classification datasets.

Chapter 9

Cross-Validation for Imbalanced Datasets

Model evaluation involves using the available dataset to fit a model and estimate its performance when making predictions on unseen examples. It is a challenging problem as both the training dataset used to fit the model and the test set used to evaluate it must be sufficiently large and representative of the underlying problem so that the resulting estimate of model performance is not too optimistic or pessimistic.

The two most common approaches used for model evaluation are the train/test split and the k -fold cross-validation procedure. Both approaches can be very effective in general, although they can result in misleading results and potentially fail when used on classification problems with a severe class imbalance. In this tutorial, you will discover how to evaluate classifier models on imbalanced datasets. After completing this tutorial, you will know:

- The challenge of evaluating classifiers on datasets using train/test splits and cross-validation.
- How a naive application of k -fold cross-validation and train-test splits will fail when evaluating classifiers on imbalanced datasets.
- How modified k -fold cross-validation and train-test splits can be used to preserve the class distribution in the dataset.

Let's get started.

9.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Challenge of Evaluating Classifiers
2. Failure of k -Fold Cross-Validation
3. Fix Cross-Validation for Imbalanced Classification

9.2 Challenge of Evaluating Classifiers

Evaluating a classification model is challenging because we won't know how good a model is until it is used. Instead, we must estimate the performance of a model using available data where we already have the target or outcome. Model evaluation involves more than just evaluating a model; it includes testing different data preparation schemes, different learning algorithms, and different hyperparameters for well-performing learning algorithms.

$$\text{Model} = \text{Data Preparation} + \text{Learning Algorithm} + \text{Hyperparameters} \quad (9.1)$$

Ideally, the model construction procedure (data preparation, learning algorithm, and hyperparameters) with the best score (with your chosen metric) can be selected and used. The simplest model evaluation procedure is to split a dataset into two parts and use one part for training a model and the second part for testing the model. As such, the parts of the dataset are named for their function, train set and test set respectively.

This is effective if your collected dataset is very large and representative of the problem. The number of examples required will differ from problem to problem, but may be thousands, hundreds of thousands, or millions of examples to be sufficient. A split of 50/50 for train and test would be ideal, although more skewed splits are common, such as 67/33 or 80/20 for train and test sets.

We rarely have enough data to get an unbiased estimate of performance using a train/test split evaluation of a model. Instead, we often have a much smaller dataset than would be preferred, and data sampling strategies must be used on this dataset. The most used model evaluation scheme for classifiers is the 10-fold cross-validation procedure.

The k -fold cross-validation procedure involves splitting the training dataset into k folds. The first $k - 1$ folds are used to train a model, and the holdout k^{th} fold is used as the test set. This process is repeated and each of the folds is given an opportunity to be used as the holdout test set. A total of k models are fit and evaluated, and the performance of the model is calculated as the mean of these runs.

The procedure has been shown to give a less optimistic estimate of model performance on small training datasets than a single train/test split. A value of $k = 10$ has been shown to be effective across a wide range of dataset sizes and model types.

9.3 Failure of k -Fold Cross-Validation

Sadly, k -fold cross-validation is not appropriate for evaluating imbalanced classifiers.

A 10-fold cross-validation, in particular, the most commonly used error-estimation method in machine learning, can easily break down in the case of class imbalances, even if the skew is less extreme than the one previously considered.

— Page 188, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

The reason is that the data is split into k -folds with a uniform probability distribution. This might work fine for data with a balanced class distribution, but when the distribution is severely skewed, it is likely that one or more folds will have few or no examples from the minority class.

This means that some or perhaps many of the model evaluations will be misleading, as the model need only predict the majority class correctly.

We can make this concrete with an example. First, we can define a dataset with a 1:100 minority to majority class distribution. This can be achieved using the `make_classification()` function for creating a synthetic dataset, specifying the number of examples (1,000), the number of classes (2), and the weighting of each class (99% and 1%).

```
...
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, weights=[0.99, 0.01], flip_y=0,
                           random_state=1)
```

Listing 9.1: Example of defining an imbalanced binary classification dataset.

The example below generates the synthetic binary classification dataset and summarizes the class distribution.

```
# create a binary classification dataset
from numpy import unique
from sklearn.datasets import make_classification
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, weights=[0.99, 0.01], flip_y=0,
                           random_state=1)
# summarize dataset
classes = unique(y)
total = len(y)
for c in classes:
    n_examples = len(y[y==c])
    percent = n_examples / total * 100
    print('> Class=%d : %d/%d (%.1f%%)' % (c, n_examples, total, percent))
```

Listing 9.2: Example of defining and summarizing the imbalanced dataset.

Running the example creates the dataset and summarizes the number of examples in each class. By setting the `random_state` argument, it ensures that we get the same randomly generated examples each time the code is run.

```
> Class=0 : 990/1000 (99.0%)
> Class=1 : 10/1000 (1.0%)
```

Listing 9.3: Example output from summarizing the imbalanced dataset.

A total of 10 examples in the minority class is not many. If we used 10-folds, we would get one example in each fold in the ideal case, which is not enough to train a model. For demonstration purposes, we will use 5-folds. In the ideal case, we would have $10/5$ or two examples in each fold, meaning 4×2 (8) folds worth of examples in a training dataset and 1×2 (2) folds in a given test dataset. First, we will use the `KFold` class to randomly split the dataset into 5-folds and check the composition of each train and test set. The complete example is listed below.

```
# example of k-fold cross-validation with an imbalanced dataset
from sklearn.datasets import make_classification
from sklearn.model_selection import KFold
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, weights=[0.99, 0.01], flip_y=0,
                           random_state=1)
```

```

kfold = KFold(n_splits=5, shuffle=True, random_state=1)
# enumerate the splits and summarize the distributions
for train_ix, test_ix in kfold.split(X):
    # select rows
    train_X, test_X = X[train_ix], X[test_ix]
    train_y, test_y = y[train_ix], y[test_ix]
    # summarize train and test composition
    train_0, train_1 = len(train_y[train_y==0]), len(train_y[train_y==1])
    test_0, test_1 = len(test_y[test_y==0]), len(test_y[test_y==1])
    print('>Train: 0=%d, 1=%d, Test: 0=%d, 1=%d' % (train_0, train_1, test_0, test_1))

```

Listing 9.4: Example of naive cross-validation on the imbalanced dataset.

Running the example creates the same dataset and enumerates each split of the data, showing the class distribution for both the train and test sets. We can see that in this case, there are some splits that have the expected 8/2 split for train and test sets, and others that are much worse, such as 6/4 (optimistic) and 10/0 (pessimistic). Evaluating a model on these splits of the data would not give a reliable estimate of performance.

```

>Train: 0=791, 1=9, Test: 0=199, 1=1
>Train: 0=793, 1=7, Test: 0=197, 1=3
>Train: 0=794, 1=6, Test: 0=196, 1=4
>Train: 0=790, 1=10, Test: 0=200, 1=0
>Train: 0=792, 1=8, Test: 0=198, 1=2

```

Listing 9.5: Example output from naive cross-validation on the imbalanced dataset.

We can demonstrate that a similar issue exists if we use a simple train/test split of the dataset, although the issue is less severe. We can use the `train_test_split()` function to create a 50/50 split of the dataset and, on average, we would expect five examples from the minority class to appear in each dataset if we performed this split many times. The complete example is listed below.

```

# example of train/test split with an imbalanced dataset
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, weights=[0.99, 0.01], flip_y=0,
                           random_state=1)
# split into train/test sets with same class ratio
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2)
# summarize
train_0, train_1 = len(trainy[trainy==0]), len(trainy[trainy==1])
test_0, test_1 = len(testy[testy==0]), len(testy[testy==1])
print('>Train: 0=%d, 1=%d, Test: 0=%d, 1=%d' % (train_0, train_1, test_0, test_1))

```

Listing 9.6: Example of naive train-test split on the imbalanced dataset.

Running the example creates the same dataset as before and splits it into a random train and test split. In this case, we can see only three examples of the minority class are present in the training set, with seven in the test set. Evaluating models on this split would not give them enough examples to learn from, too many to be evaluated on, and likely give poor performance. You can imagine how the situation could be worse with an even more severe random split.

```
>Train: 0=497, 1=3, Test: 0=493, 1=7
```

Listing 9.7: Example output from naive train-test split on the imbalanced dataset.

9.4 Fix Cross-Validation for Imbalanced Classification

The solution is to not split the data randomly when using k -fold cross-validation or a train-test split. Specifically, we can split a dataset randomly, although in such a way that maintains the same class distribution in each subset. This is called stratification or stratified sampling and the target variable (y), the class, is used to control the sampling process. For example, we can use a version of k -fold cross-validation that preserves the imbalanced class distribution in each fold. It is called stratified k -fold cross-validation and will enforce the class distribution in each split of the data to match the distribution in the complete training dataset.

... it is common, in the case of class imbalances in particular, to use stratified 10-fold cross-validation, which ensures that the proportion of positive to negative examples found in the original distribution is respected in all the folds.

— Page 205, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

We can make this concrete with an example. We can stratify the splits using the `StratifiedKFold` class that supports stratified k -fold cross-validation as its name suggests. Below is the same dataset and the same example with the stratified version of cross-validation.

```
# example of stratified k-fold cross-validation with an imbalanced dataset
from sklearn.datasets import make_classification
from sklearn.model_selection import StratifiedKFold
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, weights=[0.99, 0.01], flip_y=0,
                           random_state=1)
kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=1)
# enumerate the splits and summarize the distributions
for train_ix, test_ix in kfold.split(X, y):
    # select rows
    train_X, test_X = X[train_ix], X[test_ix]
    train_y, test_y = y[train_ix], y[test_ix]
    # summarize train and test composition
    train_0, train_1 = len(train_y[train_y==0]), len(train_y[train_y==1])
    test_0, test_1 = len(test_y[test_y==0]), len(test_y[test_y==1])
    print('>Train: 0=%d, 1=%d, Test: 0=%d, 1=%d' % (train_0, train_1, test_0, test_1))
```

Listing 9.8: Example of stratified cross-validation on the imbalanced dataset.

Running the example generates the dataset as before and summarizes the class distribution for the train and test sets for each split. In this case, we can see that each split matches what we expected in the ideal case. Each of the examples in the minority class is given one opportunity to be used in a test set, and each train and test set for each split of the data has the same class distribution.

```
>Train: 0=792, 1=8, Test: 0=198, 1=2
```

Listing 9.9: Example output from stratified cross-validation on the imbalanced dataset.

This example highlights the need to first select a value of k for k -fold cross-validation to ensure that there are a sufficient number of examples in the train and test sets to fit and evaluate a model (two examples from the minority class in the test set is probably too few for a test set).

It also highlights the requirement to use stratified k -fold cross-validation with imbalanced datasets to preserve the class distribution in the train and test sets for each evaluation of a given model. We can also use a stratified version of a train/test split. This can be achieved by setting the `stratify` argument on the call to `train_test_split()` and setting it to the `y` variable containing the target variable from the dataset. From this, the function will determine the desired class distribution and ensure that the train and test sets both have this distribution. We can demonstrate this with a worked example, listed below.

```
# example of stratified train/test split with an imbalanced dataset
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, weights=[0.99, 0.01], flip_y=0,
    random_state=1)
# split into train/test sets with same class ratio
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2,
    stratify=y)
# summarize
train_0, train_1 = len(trainy[trainy==0]), len(trainy[trainy==1])
test_0, test_1 = len(testy[testy==0]), len(testy[testy==1])
print('>Train: 0=%d, 1=%d, Test: 0=%d, 1=%d' % (train_0, train_1, test_0, test_1))
```

Listing 9.10: Example of stratified train-test split on the imbalanced dataset.

Running the example creates a random split of the dataset into training and test sets, ensuring that the class distribution is preserved, in this case leaving five examples in each dataset.

```
>Train: 0=495, 1=5, Test: 0=495, 1=5
```

Listing 9.11: Example output from stratified train-test split on the imbalanced dataset.

9.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

9.5.1 Books

- *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.
<https://amzn.to/32K9K6d>

9.5.2 API

- `sklearn.model_selection.KFold` API.
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html

- `sklearn.model_selection.StratifiedKFold` API.
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html
- `sklearn.model_selection.train_test_split` API.
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

9.6 Summary

In this tutorial, you discovered how to evaluate classifier models on imbalanced datasets. Specifically, you learned:

- The challenge of evaluating classifiers on datasets using train/test splits and cross-validation.
- How a naive application of k -fold cross-validation and train-test splits will fail when evaluating classifiers on imbalanced datasets.
- How modified k -fold cross-validation and train-test splits can be used to preserve the class distribution in the dataset.

9.6.1 Next

This was the final tutorial in this Part. In the next Part, you will discover the data sampling methods that you can use for imbalanced classification.

Part IV

Data Sampling

Chapter 10

Tour of Data Sampling Methods

Machine learning techniques often fail or give misleadingly optimistic performance on classification datasets with an imbalanced class distribution. The reason is that many machine learning algorithms are designed to operate on classification data with an equal number of observations for each class. When this is not the case, algorithms can learn that the very few minority class examples are not important and can be ignored in order to achieve good performance.

Data sampling provides a collection of techniques that transform a training dataset in order to balance or better balance the class distribution. Once balanced, standard machine learning algorithms can be trained directly on the transformed dataset without any modification. This allows the challenge of imbalanced classification, even with severely imbalanced class distributions, to be addressed with a data preparation method.

There are many different types of data sampling methods that can be used, and there is no single best method to use on all classification problems and with all classification models. Like choosing a predictive model, careful experimentation is required to discover what works best for your project. In this tutorial, you will discover a suite of data sampling techniques that can be used to balance an imbalanced classification dataset. After completing this tutorial, you will know:

- The challenge of machine learning with imbalanced classification datasets.
- The balancing of skewed class distributions using data sampling techniques.
- Tour of data sampling methods for oversampling, undersampling, and combinations of methods.

Let's get started.

10.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Problem of an Imbalanced Class Distribution
2. Balance the Class Distribution With Sampling
3. Tour of Popular Data Sampling Methods

10.2 Problem of an Imbalanced Class Distribution

Imbalanced classification involves a dataset where the class distribution is not equal. This means that the number of examples that belong to each class in the training dataset varies, often widely. It is not uncommon to have a severe skew in the class distribution, such as 1:10, 1:1000 or even 1:1000 ratio of examples in the minority class to those in the majority class.

... we define imbalanced learning as the learning process for data representation and information extraction with severe data distribution skews to develop effective decision boundaries to support the decision-making process.

— Page 1, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

Although often described in terms of two-class classification problems, class imbalance also affects those datasets with more than two classes that may have multiple minority classes or multiple majority classes. A chief problem with imbalanced classification datasets is that standard machine learning algorithms do not perform well on them. Many machine learning algorithms rely upon the class distribution in the training dataset to gauge the likelihood of observing examples in each class when the model will be used to make predictions. As such, many machine learning algorithms, like decision trees, k -nearest neighbors, and neural networks, will therefore learn that the minority class is not as important as the majority class and put more attention and perform better on the majority class.

The hitch with imbalanced datasets is that standard classification learning algorithms are often biased towards the majority classes (known as “negative”) and therefore there is a higher misclassification rate in the minority class instances (called the “positive” class).

— Page 79, *Learning from Imbalanced Data Sets*, 2018.

This is a problem because the minority class is exactly the class that we care most about in imbalanced classification problems. The reason for this is because the majority class often reflects a normal case, whereas the minority class represents a positive case for a diagnostic, fault, fraud, or other types of exceptional circumstance.

10.3 Balance the Class Distribution With Sampling

The most popular solution to an imbalanced classification problem is to change the composition of the training dataset. Techniques designed to change the class distribution in the training dataset are generally referred to as sampling methods as we are sampling an existing data sample.

Sampling methods seem to be the dominant type of approach in the community as they tackle imbalanced learning in a straightforward manner.

— Page 3, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

The reason that sampling methods are so common is because they are simple to understand and implement, and because once applied to transform the training dataset, a suite of standard machine learning algorithms can then be used directly. This means that any from tens or hundreds of machine learning algorithms developed for balanced (or mostly balanced) classification can then be fit on the training dataset without any modification adapting them for the imbalance in observations.

Basically, instead of having the model deal with the imbalance, we can attempt to balance the class frequencies. Taking this approach eliminates the fundamental imbalance issue that plagues model training.

— Page 427, *Applied Predictive Modeling*, 2013.

Machine learning algorithms like the Naive Bayes Classifier learn the likelihood of observing examples from each class from the training dataset. By fitting these models on a sampled training dataset with an artificially more equal class distribution, it allows them to learn a less biased prior probability and instead focus on the specifics (or evidence) from each input variable to discriminate the classes.

Some models use prior probabilities, such as naive Bayes and discriminant analysis classifiers. Unless specified manually, these models typically derive the value of the priors from the training data. Using more balanced priors or a balanced training set may help deal with a class imbalance.

— Page 426, *Applied Predictive Modeling*, 2013.

Sampling is only performed on the training dataset, the dataset used by an algorithm to learn a model. It is not performed on the holdout test or validation dataset. The reason is that the intent is not to remove the class bias from the model fit but to continue to evaluate the resulting model on data that is both real and representative of the target problem domain. As such, we can think of data sampling methods as addressing the problem of relative class imbalance in the training dataset, and ignoring the underlying cause of the imbalance in the problem domain. This is the difference between so-called relative and absolute rarity of examples in a minority class.

Sampling methods are a very popular method for dealing with imbalanced data. These methods are primarily employed to address the problem with relative rarity but do not address the issue of absolute rarity.

— Page 29, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

Evaluating a model on a transformed dataset with examples deleted or synthesized would likely provide a misleading and perhaps optimistic estimation of performance. There are two main types of data sampling used on the training dataset: oversampling and undersampling. In the next section, we will take a tour of popular methods from each type, as well as methods that combine multiple approaches.

10.4 Tour of Popular Data Sampling Methods

There are tens, if not hundreds, of data sampling methods to choose from in order to adjust the class distribution of the training dataset. There is no best data sampling method, just like there is no best machine learning algorithm. The methods behave differently depending on the choice of learning algorithm and on the density and composition of the training dataset.

... in many cases, sampling can mitigate the issues caused by an imbalance, but there is no clear winner among the various approaches. Also, many modeling techniques react differently to sampling, further complicating the idea of a simple guideline for which procedure to use.

— Page 429, *Applied Predictive Modeling*, 2013.

As such, it is important to carefully design experiments to test and evaluate a suite of different methods and different configurations for some methods in order to discover what works best for your specific project. Although there are many techniques to choose from, there are perhaps a dozen that are more popular and perhaps more successful on average. In this section, we will take a tour of these methods organized into a rough taxonomy of oversampling, undersampling, and combined methods.

Representative work in this area includes random oversampling, random undersampling, synthetic sampling with data generation, cluster-based sampling methods, and integration of sampling and boosting.

— Page 3, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

The following sections review some of the more popular methods, described in the context of binary (two-class) classification problems, which is a common practice, although most can be used directly or adapted for imbalanced classification with more than two classes. The list here is based mostly on the approaches available in the scikit-learn friendly library, called imbalanced-learn. For a longer list of data sampling methods, see Chapter 5 Data Level Preprocessing Methods in the 2018 book *Learning from Imbalanced Data Sets*.

10.4.1 Oversampling Techniques

Oversampling methods duplicate examples in the minority class or synthesize new examples from the examples in the minority class. Some of the more widely used and implemented oversampling methods include:

- Random Oversampling
- Synthetic Minority Oversampling Technique (SMOTE)
- Borderline-SMOTE
- Borderline Oversampling with SVM
- Adaptive Synthetic Sampling (ADASYN)

Let's take a closer look at these methods. The simplest oversampling method involves randomly duplicating examples from the minority class in the training dataset, referred to as Random Oversampling. The most popular and perhaps most successful oversampling method is SMOTE; that is an acronym for Synthetic Minority Oversampling Technique. SMOTE works by selecting examples that are close in the feature space, drawing a line between the examples in the feature space and drawing a new sample as a point along that line.

There are many extensions to the SMOTE method that aim to be more selective for the types of examples in the minority class that are synthesized. Borderline-SMOTE involves selecting those instances of the minority class that are misclassified, such as with a k -nearest neighbor classification model, and only generating synthetic samples that are *difficult* to classify. Borderline Oversampling is an extension to SMOTE that fits an SVM to the dataset and uses the decision boundary as defined by the support vectors as the basis for generating synthetic examples, again based on the idea that the decision boundary is the area where more minority examples are required.

Adaptive Synthetic Sampling (ADASYN) is another extension to SMOTE that generates synthetic samples inversely proportional to the density of the examples in the minority class. It is designed to create synthetic examples in regions of the feature space where the density of minority examples is low, and fewer or none where the density is high. For more on oversampling methods, see Chapter 12.

10.4.2 Undersampling Techniques

Undersampling methods delete or select a subset of examples from the majority class. Some of the more widely used and implemented undersampling methods include:

- Random Undersampling
- Condensed Nearest Neighbor Rule (CNN)
- Near Miss Undersampling
- Tomek Links Undersampling
- Edited Nearest Neighbors Rule (ENN)
- One-Sided Selection (OSS)
- Neighborhood Cleaning Rule (NCR)

Let's take a closer look at these methods. The simplest undersampling method involves randomly deleting examples from the majority class in the training dataset, referred to as random undersampling. One group of techniques involves selecting a robust and representative subset of the examples in the majority class. The Condensed Nearest Neighbors rule, or CNN for short, was designed for reducing the memory required for the k -nearest neighbors algorithm. It works by enumerating the examples in the dataset and adding them to the store only if they cannot be classified correctly by the current contents of the store, and can be applied to reduce the number of examples in the majority class after all examples in the minority class have been added to the store.

Near Miss refers to a family of methods that use KNN to select examples from the majority class. NearMiss-1 selects examples from the majority class that have the smallest average distance to the three closest examples from the minority class. NearMiss-2 selects examples from the majority class that have the smallest average distance to the three furthest examples from the minority class. NearMiss-3 involves selecting a given number of majority class examples for each example in the minority class that are closest.

Another group of techniques involves selecting examples from the majority class to delete. These approaches typically involve identifying those examples that are challenging to classify and therefore add ambiguity to the decision boundary. Perhaps the most widely known deletion undersampling approach is referred to as Tomek Links, originally developed as part of an extension to the Condensed Nearest Neighbors rule. A Tomek Link refers to a pair of examples in the training dataset that are both nearest neighbors (have the minimum distance in feature space) and belong to different classes. Tomek Links are often misclassified examples found along the class boundary and the examples in the majority class are deleted.

The Edited Nearest Neighbors rule, or ENN for short, is another method for selecting examples for deletion. This rule involves using $k = 3$ nearest neighbors to locate those examples in a dataset that are misclassified and deleting them. The ENN procedure can be repeated multiple times on the same dataset, better refining the selection of examples in the majority class. This extension is referred to initially as *unlimited editing* although it is more commonly referred to as Repeatedly Edited Nearest Neighbors. Staying with the *select to keep* vs. *select to delete* families of undersampling methods, there are also undersampling methods that combine both approaches.

One-Sided Selection, or OSS for short, is an undersampling technique combines Tomek Links and the Condensed Nearest Neighbor (CNN) Rule. The Tomek Links method is used to remove noisy examples on the class boundary, whereas CNN is used to remove redundant examples from the interior of the density of the majority class. The Neighborhood Cleaning Rule, or NCR for short, is another combination undersampling technique that combines both the Condensed Nearest Neighbor (CNN) Rule to remove redundant examples and the Edited Nearest Neighbors (ENN) Rule to remove noisy or ambiguous examples. For more on oversampling methods, see Chapter 13.

10.4.3 Combinations of Techniques

Although an oversampling or undersampling method when used alone on a training dataset can be effective, experiments have shown that applying both types of techniques together can often result in better overall performance of a model fit on the resulting transformed dataset. Some of the more widely used and implemented combinations of data sampling methods include:

- SMOTE and Random Undersampling
- SMOTE and Tomek Links
- SMOTE and Edited Nearest Neighbors Rule

Let's take a closer look at these methods. SMOTE is perhaps the most popular and widely used oversampling technique. It is common to pair SMOTE with an undersampling method that selects examples from the dataset to delete, and the procedure is applied to the dataset

after SMOTE, allowing the editing step to be applied to both the minority and majority class. The intent is to remove noisy points along the class boundary from both classes, which seems to have the effect of the better performance of classifiers fit on the transformed dataset.

Two popular examples involve using SMOTE followed by the deletion of Tomek Links, and SMOTE followed by the deletion of those examples misclassified via a KNN model, the so-called Edited Nearest Neighbors rule. For more on combining oversampling and undersampling methods, see Chapter 14.

10.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

10.5.1 Papers

- *SMOTE: Synthetic Minority Over-sampling Technique*, 2011.
<https://arxiv.org/abs/1106.1813>
- *A Study of the Behavior of Several Methods for Balancing Machine Learning Training Data*, 2004.
<https://dl.acm.org/citation.cfm?id=1007735>

10.5.2 Books

- *Applied Predictive Modeling*, 2013.
<https://amzn.to/2VRASxV>
- *Learning from Imbalanced Data Sets*, 2018.
<https://amzn.to/307Xlva>
- *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.
<https://amzn.to/32K9K6d>

10.5.3 Articles

- Oversampling and undersampling in data analysis, Wikipedia.
https://en.wikipedia.org/wiki/Oversampling_and_undersampling_in_data_analysis

10.6 Summary

In this tutorial, you discovered a suite of data sampling techniques that can be used to balance an imbalanced classification dataset. Specifically, you learned:

- The challenge of machine learning with imbalanced classification datasets.
- The balancing of skewed class distributions using data sampling techniques.
- Tour of popular data sampling methods for oversampling, undersampling, and combinations of methods.

10.6.1 Next

In the next tutorial, you will discover how to use random undersampling and random oversampling to change the class distribution of training datasets.

Chapter 11

Random Data Sampling

Imbalanced datasets are those where there is a severe skew in the class distribution, such as 1:100 or 1:1000 examples in the minority class to the majority class. This bias in the training dataset can influence many machine learning algorithms, leading some to ignore the minority class entirely. This is a problem as it is typically the minority class on which predictions are most important.

One approach to addressing the problem of class imbalance is to randomly sample the training dataset. The two main approaches to randomly sampling an imbalanced dataset are to delete examples from the majority class, called undersampling, and to duplicate examples from the minority class, called oversampling. In this tutorial, you will discover random oversampling and undersampling for imbalanced classification. After completing this tutorial, you will know:

- Random sampling provides a naive technique for rebalancing the class distribution for an imbalanced dataset.
- Random oversampling duplicates examples from the minority class in the training dataset and can result in overfitting for some models.
- Random undersampling deletes examples from the majority class and can result in losing information invaluable to a model.

Let's get started.

Note: This chapter makes use of the imbalanced-learn library. See Appendix [B](#) for installation instructions, if needed.

11.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Random Sampling
2. Random Oversampling
3. Random Undersampling

11.2 Random Sampling

Data sampling involves creating a new transformed version of the training dataset in which the selected examples have a different class distribution. This is a simple and effective strategy for imbalanced classification problems.

Applying re-sampling strategies to obtain a more balanced data distribution is an effective solution to the imbalance problem

— *A Survey of Predictive Modelling under Imbalanced Distributions*, 2015.

The simplest strategy is to choose examples for the transformed dataset randomly, called random sampling. There are two main approaches to random sampling for imbalanced classification; they are oversampling and undersampling.

- **Random Oversampling:** Randomly duplicate examples in the minority class.
- **Random Undersampling:** Randomly delete examples in the majority class.

Random oversampling involves randomly selecting examples from the minority class, with replacement, and adding them to the training dataset. Random undersampling involves randomly selecting examples from the majority class and deleting them from the training dataset.

In the random under-sampling, the majority class instances are discarded at random until a more balanced distribution is reached.

— Page 45, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

Both approaches can be repeated until the desired class distribution is achieved in the training dataset, such as an equal split across the classes. They are referred to as *naive sampling* methods because they assume nothing about the data and no heuristics are used. This makes them simple to implement and fast to execute, which is desirable for very large and complex datasets.

Both techniques can be used for two-class (binary) classification problems and multiclass classification problems with one or more majority or minority classes. Importantly, the change to the class distribution is only applied to the training dataset. The intent is to influence the fit of the models. The sampling is not applied to the test or holdout dataset used to evaluate the performance of a model.

Generally, these naive methods can be effective, although that depends on the specifics of the dataset and models involved. Let's take a closer look at each method and how to use them in practice.

11.3 Random Oversampling

Random oversampling involves randomly duplicating examples from the minority class and adding them to the training dataset. Examples from the training dataset are selected randomly with replacement. This means that examples from the minority class can be chosen and added to the new *more balanced* training dataset multiple times; they are selected from the original

training dataset, added to the new training dataset, and then returned or *replaced* in the original dataset, allowing them to be selected again.

This technique can be effective for those machine learning algorithms that are affected by a skewed distribution and where multiple duplicate examples for a given class can influence the fit of the model. This might include algorithms that iteratively learn coefficients, like artificial neural networks that use stochastic gradient descent. It can also affect models that seek good splits of the data, such as support vector machines and decision trees.

It might be useful to tune the target class distribution. In some cases, seeking a balanced distribution for a severely imbalanced dataset can cause affected algorithms to overfit the minority class, leading to increased generalization error. The effect can be better performance on the training dataset, but worse performance on the holdout or test dataset.

... the random oversampling may increase the likelihood of occurring overfitting, since it makes exact copies of the minority class examples. In this way, a symbolic classifier, for instance, might construct rules that are apparently accurate, but actually cover one replicated example.

— Page 83, *Learning from Imbalanced Data Sets*, 2018.

As such, to gain insight into the impact of the method, it is a good idea to monitor the performance on both train and test datasets after oversampling and compare the results to the same algorithm on the original dataset. The increase in the number of examples for the minority class, especially if the class skew was severe, can also result in a marked increase in the computational cost when fitting the model, especially considering the model is seeing the same examples in the training dataset again and again.

... in random over-sampling, a random set of copies of minority class examples is added to the data. This may increase the likelihood of overfitting, specially for higher over-sampling rates. Moreover, it may decrease the classifier performance and increase the computational effort.

— *A Survey of Predictive Modelling under Imbalanced Distributions*, 2015.

Random oversampling can be implemented using the `RandomOverSampler` class. The class can be defined and takes a `sampling_strategy` argument that can be set to '`minority`' to automatically balance the minority class with majority class or classes. For example:

```
...
# define oversampling strategy
oversample = RandomOverSampler(sampling_strategy='minority')
```

Listing 11.1: Example of defining the random oversampling strategy.

This means that if the majority class had 1,000 examples and the minority class had 100, this strategy would oversampling the minority class so that it has 1,000 examples. A floating point value can be specified to indicate the desired ratio of minority to majority class examples in the transformed dataset. For example:

```
...
# define oversampling strategy
oversample = RandomOverSampler(sampling_strategy=0.5)
```

Listing 11.2: Example of defining the random oversampling strategy to balanced to 50% of the majority class.

This would ensure that the minority class was oversampled to have half the number of examples as the majority class, for binary classification problems. This means that if the majority class had 1,000 examples and the minority class had 100, the transformed dataset would have 500 examples of the minority class. The class is like a scikit-learn transform object in that it is fit on a dataset, then used to generate a new or transformed dataset. Unlike the scikit-learn transforms, it will change the number of examples in the dataset, not just the values (like a scaler) or number of features (like a projection). For example, it can be fit and applied in one step by calling the `fit_sample()` function:

```
...
# fit and apply the transform
X_over, y_over = oversample.fit_resample(X, y)
```

Listing 11.3: Example of fitting the oversampling strategy.

We can demonstrate this on a simple synthetic binary classification problem with a 1:100 class imbalance.

```
...
# define dataset
X, y = make_classification(n_samples=10000, weights=[0.99], flip_y=0)
```

Listing 11.4: Example of defining an imbalanced classification dataset.

The complete example of defining the dataset and performing random oversampling to balance the class distribution is listed below.

```
# example of random oversampling to balance the class distribution
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.over_sampling import RandomOverSampler
# define dataset
X, y = make_classification(n_samples=10000, weights=[0.99], flip_y=0)
# summarize class distribution
print(Counter(y))
# define oversampling strategy
oversample = RandomOverSampler(sampling_strategy='minority')
# fit and apply the transform
X_over, y_over = oversample.fit_resample(X, y)
# summarize class distribution
print(Counter(y_over))
```

Listing 11.5: Example of random oversampling the minority class.

Running the example first creates the dataset, then summarizes the class distribution. We can see that there are nearly 10K examples in the majority class and 100 examples in the minority class. Then the random oversample transform is defined to balance the minority class, then fit and applied to the dataset. The class distribution for the transformed dataset is reported showing that now the minority class has the same number of examples as the majority class.

```
Counter({0: 9900, 1: 100})
Counter({0: 9900, 1: 9900})
```

Listing 11.6: Example output from random oversampling the minority class.

This transform can be used as part of a `Pipeline` to ensure that it is only applied to the training dataset as part of each split in a k -fold cross-validation. A traditional scikit-learn `Pipeline` cannot be used; instead, a `Pipeline` from the imbalanced-learn library can be used. For example:

```
...
# pipeline
steps = [('over', RandomOverSampler()), ('model', DecisionTreeClassifier())]
pipeline = Pipeline(steps=steps)
```

Listing 11.7: Example of defining a `Pipeline` with oversampling and a model.

The example below provides a complete example of evaluating a decision tree on an imbalanced dataset with a 1:100 class distribution. The model is evaluated using repeated 10-fold cross-validation with three repeats, and the oversampling is performed on the training dataset within each fold separately, ensuring that there is no data leakage as might occur if the oversampling was performed prior to the cross-validation. Data leakage refers to using information from the test dataset when fitting the model and can result in an optimistically biased estimate of model performance.

```
# example of evaluating a decision tree with random oversampling
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.tree import DecisionTreeClassifier
from imblearn.pipeline import Pipeline
from imblearn.over_sampling import RandomOverSampler
# define dataset
X, y = make_classification(n_samples=10000, weights=[0.99], flip_y=0)
# define pipeline
steps = [('over', RandomOverSampler()), ('model', DecisionTreeClassifier())]
pipeline = Pipeline(steps=steps)
# evaluate pipeline
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
scores = cross_val_score(pipeline, X, y, scoring='f1_micro', cv=cv, n_jobs=-1)
score = mean(scores)
print('F-measure: %.3f' % score)
```

Listing 11.8: Example of random oversampling during model evaluation.

Running the example evaluates the decision tree model on the imbalanced dataset with oversampling. The chosen model and sampling configuration are arbitrary, designed to provide a template that you can use to test oversampling with your dataset and learning algorithm, rather than optimally solve the synthetic dataset. The default oversampling strategy is used, which balances the minority classes with the majority class. The F-measure averaged across each fold and each repeat is reported. Your specific results may differ given the stochastic nature of the dataset and the sampling strategy.

```
F-measure: 0.990
```

Listing 11.9: Example output from random oversampling during model evaluation.

Now that we are familiar with oversampling, let's take a look at undersampling.

11.4 Random Undersampling

Random undersampling involves randomly selecting examples from the majority class to delete from the training dataset. This has the effect of reducing the number of examples in the majority class in the transformed version of the training dataset. This process can be repeated until the desired class distribution is achieved, such as an equal number of examples for each class.

This approach may be more suitable for those datasets where there is a class imbalance although still a sufficient number of examples in the minority class, such that a useful model can be fit. A limitation of undersampling is that examples from the majority class are deleted that may be useful, important, or perhaps critical to fitting a robust decision boundary. Given that examples are deleted randomly, there is no way to detect or preserve *good* or more information-rich examples from the majority class.

... in random under-sampling (potentially), vast quantities of data are discarded.

[...] This can be highly problematic, as the loss of such data can make the decision boundary between minority and majority instances harder to learn, resulting in a loss in classification performance.

— Page 45, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

The random undersampling technique can be implemented using the `RandomUnderSampler` imbalanced-learn class. The class can be used just like the `RandomOverSampler` class in the previous section, except the strategies impact the majority class instead of the minority class. For example, setting the `sampling_strategy` argument to '`majority`' will undersample the majority class determined by the class with the largest number of examples.

```
...
# define undersample strategy
undersample = RandomUnderSampler(sampling_strategy='majority')
```

Listing 11.10: Example of defining the random undersampling strategy.

For example, a dataset with 1,000 examples in the majority class and 100 examples in the minority class will be undersampled such that both classes would have 100 examples in the transformed training dataset. We can also set the `sampling_strategy` argument to a floating point value which will be a percentage relative to the minority class, specifically the number of examples in the minority class divided by the number of examples in the majority class. If we set `sampling_strategy` to 0.5 in an imbalanced data dataset with 1,000 examples in the majority class and 100 examples in the minority class, then there would be 200 examples for the majority class in the transformed dataset (or $\frac{100}{200} = 0.5$).

```
...
# define undersample strategy
undersample = RandomUnderSampler(sampling_strategy=0.5)
```

Listing 11.11: Example of defining the random undersampling strategy to be larger than the minority class.

This might be preferred to ensure that the resulting dataset is both large enough to fit a reasonable model, and that not too much useful information from the majority class is discarded. The transform can then be fit and applied to a dataset in one step by calling the `fit_resample()` function and passing the untransformed dataset as arguments.

```
...
# fit and apply the transform
X_over, y_over = undersample.fit_resample(X, y)
```

Listing 11.12: Example of fitting the undersampling strategy.

We can demonstrate this on a dataset with a 1:100 class imbalance. The complete example is listed below.

```
# example of random undersampling to balance the class distribution
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.under_sampling import RandomUnderSampler
# define dataset
X, y = make_classification(n_samples=10000, weights=[0.99], flip_y=0)
# summarize class distribution
print(Counter(y))
# define undersample strategy
undersample = RandomUnderSampler(sampling_strategy='majority')
# fit and apply the transform
X_over, y_over = undersample.fit_resample(X, y)
# summarize class distribution
print(Counter(y_over))
```

Listing 11.13: Example of random undersampling the majority class.

Running the example first creates the dataset and reports the imbalanced class distribution. The transform is fit and applied on the dataset and the new class distribution is reported. We can see that majority class is undersampled to have the same number of examples as the minority class. Judgment and empirical results will have to be used as to whether a training dataset with just 200 examples would be sufficient to train a model.

```
Counter({0: 9900, 1: 100})
Counter({0: 100, 1: 100})
```

Listing 11.14: Example output from random undersampling the majority class.

This undersampling transform can also be used in a `Pipeline`, like in the oversampling transform from the previous section. This allows the transform to be applied to the training dataset only using evaluation schemes such as k -fold cross-validation, avoiding any data leakage in the evaluation of a model.

```
...
# define pipeline
steps = [('under', RandomUnderSampler()), ('model', DecisionTreeClassifier())]
pipeline = Pipeline(steps=steps)
```

Listing 11.15: Example of defining a `Pipeline` with undersampling and a model.

We can define an example of fitting a decision tree on an imbalanced classification dataset with the undersampling transform applied to the training dataset on each split of a repeated 10-fold cross-validation. The complete example is listed below.

```
# example of evaluating a decision tree with random undersampling
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.tree import DecisionTreeClassifier
from imblearn.pipeline import Pipeline
from imblearn.under_sampling import RandomUnderSampler
# define dataset
X, y = make_classification(n_samples=10000, weights=[0.99], flip_y=0)
# define pipeline
steps = [('under', RandomUnderSampler()), ('model', DecisionTreeClassifier())]
pipeline = Pipeline(steps=steps)
# evaluate pipeline
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
scores = cross_val_score(pipeline, X, y, scoring='f1_micro', cv=cv, n_jobs=-1)
score = mean(scores)
print('F-measure: %.3f' % score)
```

Listing 11.16: Example of random undersampling during model evaluation.

Running the example evaluates the decision tree model on the imbalanced dataset with undersampling. The chosen model and sampling configuration are arbitrary, designed to provide a template that you can use to test undersampling with your dataset and learning algorithm rather than optimally solve the synthetic dataset. The default undersampling strategy is used, which balances the majority classes with the minority class. The F-measure averaged across each fold and each repeat is reported. Your specific results may differ given the stochastic nature of the dataset and the sampling strategy.

```
F-measure: 0.889
```

Listing 11.17: Example output from random undersampling during model evaluation.

11.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

11.5.1 Papers

- *A Study Of The Behavior Of Several Methods For Balancing Machine Learning Training Data*, 2004.
<https://dl.acm.org/citation.cfm?id=1007735>
- *A Survey of Predictive Modelling under Imbalanced Distributions*, 2015.
<https://arxiv.org/abs/1505.01658>

11.5.2 Books

- Chapter 5 Data Level Preprocessing Methods, *Learning from Imbalanced Data Sets*, 2018.
<https://amzn.to/307Xlva>
- Chapter 3 Imbalanced Datasets: From Sampling to Classifiers, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.
<https://amzn.to/32K9K6d>

11.5.3 API

- Imbalanced-Learn Documentation.
<https://imbalanced-learn.org/stable/>
- imbalanced-learn, GitHub.
<https://github.com/scikit-learn-contrib/imbalanced-learn>
- imblearn.over_sampling.RandomOverSampler API.
https://imbalanced-learn.org/stable/generated/imblearn.over_sampling.RandomOverSampler.html
- imblearn.under_sampling.RandomUnderSampler API.
https://imbalanced-learn.org/stable/generated/imblearn.under_sampling.RandomUnderSampler.html

11.5.4 Articles

- Oversampling and undersampling in data analysis, Wikipedia.
https://en.wikipedia.org/wiki/Oversampling_and_undersampling_in_data_analysis

11.6 Summary

In this tutorial, you discovered random oversampling and undersampling for imbalanced classification. Specifically, you learned:

- Random sampling provides a naive technique for rebalancing the class distribution for an imbalanced dataset.
- Random oversampling duplicates examples from the minority class in the training dataset and can result in overfitting for some models.
- Random undersampling deletes examples from the majority class and can result in losing information invaluable to a model.

11.6.1 Next

In the next tutorial, you will discover how to use SMOTE oversampling to change the class distribution of training datasets.

Chapter 12

Oversampling Methods

Imbalanced classification involves developing predictive models on classification datasets that have a severe class imbalance. The challenge of working with imbalanced datasets is that most machine learning techniques will ignore, and in turn have poor performance on, the minority class, although typically it is performance on the minority class that is most important.

One approach to addressing imbalanced datasets is to oversample the minority class. The simplest approach involves duplicating examples in the minority class, although these examples don't add any new information to the model. Instead, new examples can be synthesized from the existing examples. This is a type of data augmentation for the minority class and is referred to as the Synthetic Minority Oversampling Technique, or SMOTE for short. In this tutorial, you will discover the SMOTE for oversampling imbalanced classification datasets. After completing this tutorial, you will know:

- How the SMOTE synthesizes new examples for the minority class.
- How to correctly fit and evaluate machine learning models on SMOTE-transformed training datasets.
- How to use extensions of the SMOTE that generate synthetic examples along the class decision boundary.

Let's get started.

Note: This chapter makes use of the imbalanced-learn library. See Appendix B for installation instructions, if needed.

12.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Synthetic Minority Oversampling Technique
2. SMOTE for Balancing Data
3. SMOTE for Classification
4. SMOTE With Selective Sample Generation

12.2 Synthetic Minority Oversampling Technique

A problem with imbalanced classification is that there are too few examples of the minority class for a model to effectively learn the decision boundary. One way to solve this problem is to oversample the examples in the minority class. This can be achieved by simply duplicating examples from the minority class in the training dataset prior to fitting a model. This can balance the class distribution but does not provide any additional information to the model. An improvement on duplicating examples from the minority class is to synthesize new examples from the minority class. This is a type of data augmentation for tabular data and can be very effective.

Perhaps the most widely used approach to synthesizing new examples is called the Synthetic Minority Oversampling TEchnique, or SMOTE for short. This technique was described by Nitesh Chawla, et al. in their 2002 paper named for the technique titled *SMOTE: Synthetic Minority Over-sampling Technique*. SMOTE works by selecting examples that are close in the feature space, drawing a line between the examples in the feature space and drawing a new sample at a point along that line.

Specifically, a random example from the minority class is first chosen. Then k of the nearest neighbors for that example are found (typically $k = 5$). A randomly selected neighbor is chosen and a synthetic example is created at a randomly selected point between the two examples in feature space.

... SMOTE first selects a minority class instance a at random and finds its k nearest minority class neighbors. The synthetic instance is then created by choosing one of the k nearest neighbors b at random and connecting a and b to form a line segment in the feature space. The synthetic instances are generated as a convex combination of the two chosen instances a and b .

— Page 47, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

This procedure can be used to create as many synthetic examples for the minority class as are required. As described in the paper, it suggests first using random undersampling to trim the number of examples in the majority class, then use SMOTE to oversample the minority class to balance the class distribution.

The combination of SMOTE and under-sampling performs better than plain under-sampling.

— *SMOTE: Synthetic Minority Over-sampling Technique*, 2011.

The approach is effective because new synthetic examples from the minority class are created that are plausible, that is, are relatively close in feature space to existing examples from the minority class.

Our method of synthetic over-sampling works to cause the classifier to build larger decision regions that contain nearby minority class points.

— *SMOTE: Synthetic Minority Over-sampling Technique*, 2011.

A general downside of the approach is that synthetic examples are created without considering the majority class, possibly resulting in ambiguous examples if there is a strong overlap for the classes. Now that we are familiar with the technique, let's look at a worked example for an imbalanced classification problem.

12.3 SMOTE for Balancing Data

In this section, we will develop an intuition for the SMOTE by applying it to an imbalanced binary classification problem. First, we can use the `make_classification()` scikit-learn function to create a synthetic binary classification dataset with 10,000 examples and a 1:100 class distribution.

```
...
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
```

Listing 12.1: Example of defining an imbalanced binary classification problem.

We can use the `Counter` object to summarize the number of examples in each class to confirm the dataset was created correctly.

```
...
# summarize class distribution
counter = Counter(y)
print(counter)
```

Listing 12.2: Example of summarizing the class distribution.

Finally, we can create a scatter plot of the dataset and color the examples for each class a different color to clearly see the spatial nature of the class imbalance.

```
...
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Listing 12.3: Example of creating a scatter plot with dots colored by class label.

Tying this all together, the complete example of generating and plotting a synthetic binary classification problem is listed below.

```
# Generate and plot a synthetic imbalanced classification dataset
from collections import Counter
from sklearn.datasets import make_classification
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# summarize class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Listing 12.4: Example of defining and summarizing an imbalanced classification dataset.

Running the example first summarizes the class distribution, confirms the 1:100 ratio, in this case with 9,900 examples in the majority class and 100 in the minority class.

```
Counter({0: 9900, 1: 100})
```

Listing 12.5: Example output from defining and summarizing an imbalanced classification dataset.

A scatter plot of the dataset is created showing the large mass of points that belong to the majority class (blue) and a small number of points spread out for the minority class (orange). We can see some measure of overlap between the two classes.

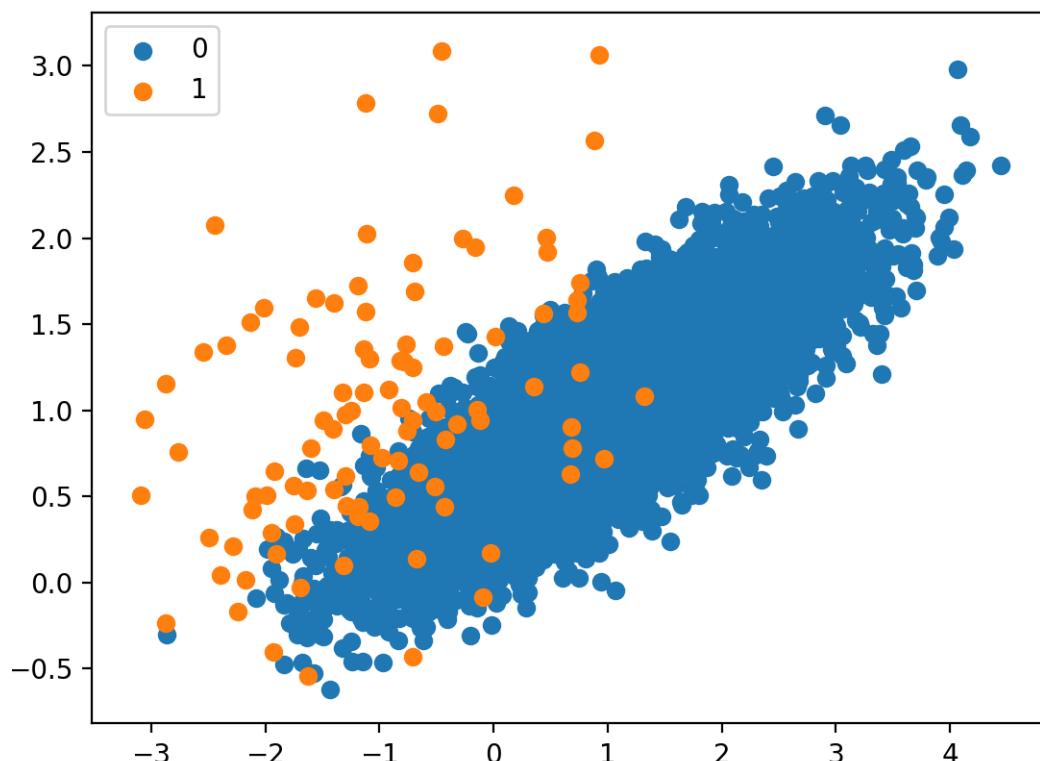


Figure 12.1: Scatter Plot of Imbalanced Binary Classification Problem.

Next, we can oversample the minority class using SMOTE and plot the transformed dataset. We can use the SMOTE implementation provided by the imbalanced-learn Python library in the `SMOTE` class. The `SMOTE` class acts like a data transform object from scikit-learn in that it must be defined and configured, fit on a dataset, then applied to create a new transformed version of the dataset. For example, we can define a `SMOTE` instance with default parameters that will balance the minority class and then fit and apply it in one step to create a transformed version of our dataset.

```
...
# transform the dataset
```

```
oversample = SMOTE()
X, y = oversample.fit_resample(X, y)
```

Listing 12.6: Example of defining and fitting SMOTE on a dataset.

Once transformed, we can summarize the class distribution of the new transformed dataset, which would expect to now be balanced through the creation of many new synthetic examples in the minority class.

```
...
# summarize the new class distribution
counter = Counter(y)
print(counter)
```

Listing 12.7: Example of summarizing the class distribution.

A scatter plot of the transformed dataset can also be created and we would expect to see many more examples for the minority class on lines between the original examples in the minority class. Tying this together, the complete examples of applying SMOTE to the synthetic dataset and then summarizing and plotting the transformed result is listed below.

```
# Oversample and plot imbalanced dataset with SMOTE
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.over_sampling import SMOTE
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# summarize class distribution
counter = Counter(y)
print(counter)
# transform the dataset
oversample = SMOTE()
X, y = oversample.fit_resample(X, y)
# summarize the new class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Listing 12.8: Example of applying SMOTE to an imbalanced classification dataset.

Running the example first creates the dataset and summarizes the class distribution, showing the 1:100 ratio. Then the dataset is transformed using the SMOTE and the new class distribution is summarized, showing a balanced distribution now with 9,900 examples in the minority class.

```
Counter({0: 9900, 1: 100})
Counter({0: 9900, 1: 9900})
```

Listing 12.9: Example output from applying SMOTE to an imbalanced classification dataset.

Finally, a scatter plot of the transformed dataset is created. It shows many more examples in the minority class created along the lines between the original examples in the minority class.

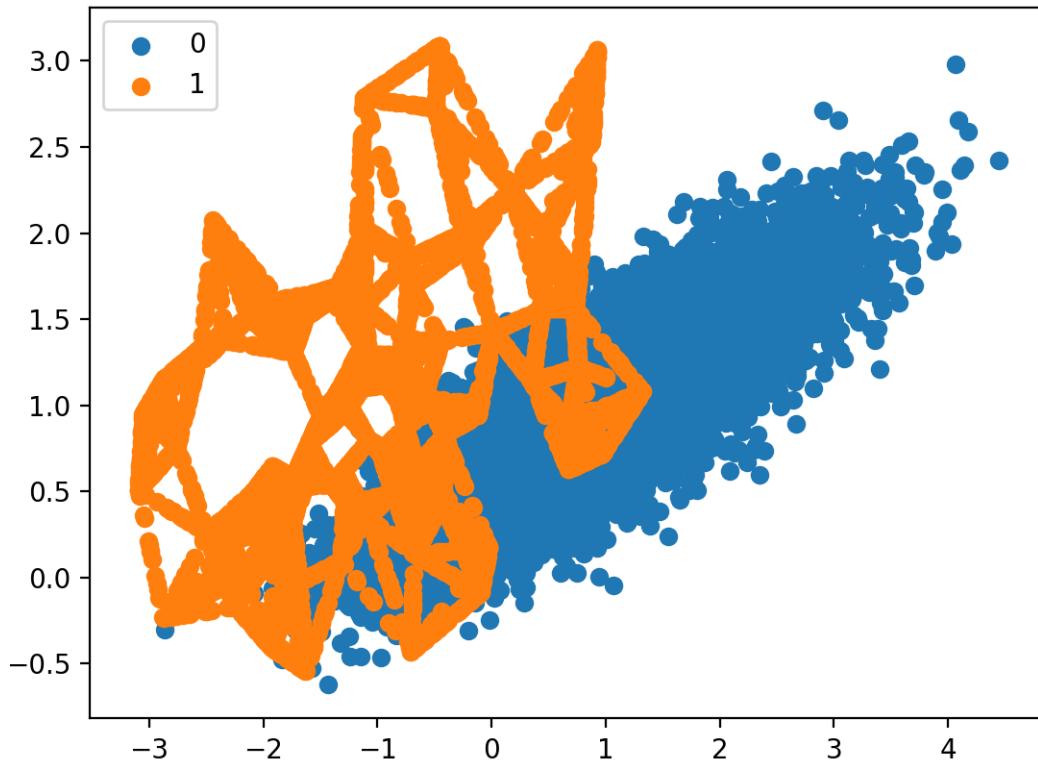


Figure 12.2: Scatter Plot of Imbalanced Binary Classification Problem Transformed by SMOTE.

Now that we are familiar with transforming imbalanced datasets, let's look at using SMOTE when fitting and evaluating classification models.

12.4 SMOTE for Classification

In this section, we will look at how we can use SMOTE as a data preparation method when fitting and evaluating machine learning algorithms in scikit-learn. First, we use our binary classification dataset from the previous section then fit and evaluate a decision tree algorithm. The algorithm is defined with any required hyperparameters (we will use the defaults), then we will use repeated stratified k -fold cross-validation to evaluate the model. We will use three repeats of 10-fold cross-validation, meaning that 10-fold cross-validation is applied three times fitting and evaluating 30 models on the dataset.

The dataset is stratified, meaning that each fold of the cross-validation split will have the same class distribution as the original dataset, in this case, a 1:100 ratio. We will evaluate the model using the ROC area under curve (AUC) metric. This can be optimistic for severely imbalanced datasets but will still show a relative change with better performing models.

```
...
# define model
model = DecisionTreeClassifier()
# evaluate pipeline
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
```

Listing 12.10: Example defining a model and model evaluation procedure.

Once fit, we can calculate and report the mean of the scores across the folds and repeats.

```
...
print('Mean ROC AUC: %.3f' % mean(scores))
```

Listing 12.11: Example of summarizing model performance.

We would not expect a decision tree fit on the raw imbalanced dataset to perform very well. Tying this together, the complete example is listed below.

```
# decision tree evaluated on imbalanced dataset
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.tree import DecisionTreeClassifier
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# define model
model = DecisionTreeClassifier()
# evaluate pipeline
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
print('Mean ROC AUC: %.3f' % mean(scores))
```

Listing 12.12: Example of evaluating a model on the imbalanced classification dataset.

Running the example evaluates the model and reports the mean ROC AUC.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that a ROC AUC of about 0.77 is reported.

```
Mean ROC AUC: 0.769
```

Listing 12.13: Example output from evaluating a model on the imbalanced classification dataset.

Now, we can try the same model and the same evaluation method, although use a SMOTE transformed version of the dataset. The correct application of oversampling during k -fold cross-validation is to apply the method to the training dataset only, then evaluate the model on the stratified but non-transformed test set. This can be achieved by defining a Pipeline that first transforms the training dataset with SMOTE then fits the model.

```
...
# define pipeline
steps = [('over', SMOTE()), ('model', DecisionTreeClassifier())]
```

```
pipeline = Pipeline(steps=steps)
```

Listing 12.14: Example of defining a Pipeline with SMOTE and a model.

This pipeline can then be evaluated using repeated k -fold cross-validation. Tying this together, the complete example of evaluating a decision tree with SMOTE oversampling on the training dataset is listed below.

```
# decision tree evaluated on imbalanced dataset with SMOTE oversampling
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.tree import DecisionTreeClassifier
from imblearn.pipeline import Pipeline
from imblearn.over_sampling import SMOTE
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# define pipeline
steps = [('over', SMOTE()), ('model', DecisionTreeClassifier())]
pipeline = Pipeline(steps=steps)
# evaluate pipeline
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
scores = cross_val_score(pipeline, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
print('Mean ROC AUC: %.3f' % mean(scores))
```

Listing 12.15: Example of evaluating a model with SMOTE on the imbalanced classification dataset.

Running the example evaluates the model and reports the mean ROC AUC score across the multiple folds and repeats.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see a modest improvement in performance from a ROC AUC of about 0.77 to about 0.83.

```
Mean ROC AUC: 0.826
```

Listing 12.16: Example output from evaluating a model with SMOTE on the imbalanced classification dataset.

You could explore testing different ratios of the minority class and majority class (e.g. changing the `sampling_strategy` argument) to see if a further lift in performance is possible. Another area to explore would be to test different values of the k -nearest neighbors selected in the SMOTE procedure when each new synthetic example is created. The default is $k = 5$, although larger or smaller values will influence the types of examples created, and in turn, may impact the performance of the model. For example, we could grid search a range of values of k , such as values from 1 to 7, and evaluate the pipeline for each value.

```
...
# values to evaluate
k_values = [1, 2, 3, 4, 5, 6, 7]
```

```

for k in k_values:
    # define pipeline
    ...

```

Listing 12.17: Example of grid searching different k -values for SMOTE.

The complete example is listed below.

```

# grid search k value for SMOTE oversampling for imbalanced classification
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.tree import DecisionTreeClassifier
from imblearn.pipeline import Pipeline
from imblearn.over_sampling import SMOTE
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# values to evaluate
k_values = [1, 2, 3, 4, 5, 6, 7]
for k in k_values:
    # define pipeline
    model = DecisionTreeClassifier()
    over = SMOTE(sampling_strategy=0.1, k_neighbors=k)
    pipeline = Pipeline(steps=[('over', over), ('model', model)])
    # evaluate pipeline
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    scores = cross_val_score(pipeline, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
    score = mean(scores)
    print('> k=%d, Mean ROC AUC: %.3f' % (k, score))

```

Listing 12.18: Example of grid searching k -values for SMOTE.

Running the example will perform SMOTE oversampling with different k values for the KNN used in the procedure, followed by fitting a decision tree on the resulting training dataset. The mean ROC AUC is reported for each configuration.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, the results suggest that a $k = 5$ might be good with a ROC AUC of about 0.81, and $k = 7$ might also be good with a ROC AUC of about 0.82. This highlights that both the amount of oversampling and undersampling performed (`sampling_strategy` argument) and the number of examples selected from which a neighbor is chosen to create a synthetic example (`k_neighbors`) may be important parameters to select and tune for your dataset.

```

> k=1, Mean ROC AUC: 0.784
> k=2, Mean ROC AUC: 0.789
> k=3, Mean ROC AUC: 0.803
> k=4, Mean ROC AUC: 0.813
> k=5, Mean ROC AUC: 0.817
> k=6, Mean ROC AUC: 0.813
> k=7, Mean ROC AUC: 0.824

```

Listing 12.19: Example output grid searching k -values for SMOTE.

Now that we are familiar with how to use SMOTE when fitting and evaluating classification models, let's look at some extensions of the SMOTE procedure.

12.5 SMOTE With Selective Sample Generation

We can be selective about the examples in the minority class that are oversampled using SMOTE. In this section, we will review some extensions to SMOTE that are more selective regarding the examples from the minority class that provide the basis for generating new synthetic examples.

12.5.1 Borderline-SMOTE

A popular extension to SMOTE involves selecting those instances of the minority class that are misclassified, such as with a k -nearest neighbor classification model. We can then oversample just those difficult instances, providing more resolution only where it may be required.

The examples on the borderline and the ones nearby [...] are more apt to be misclassified than the ones far from the borderline, and thus more important for classification.

— *Borderline-SMOTE: A New Over-Sampling Method in Imbalanced Data Sets Learning*, 2005.

These examples that are misclassified are likely ambiguous and on the edge or border of the decision boundary where class membership may overlap. As such, this modified to SMOTE is called Borderline-SMOTE and was proposed by Hui Han, et al. in their 2005 paper titled *Borderline-SMOTE: A New Over-Sampling Method in Imbalanced Data Sets Learning*. The authors also describe a version of the method that also oversampled the majority class for those examples that cause a misclassification of borderline instances in the minority class. This is referred to as Borderline-SMOTE1, whereas the oversampling of just the borderline cases in minority class is referred to as Borderline-SMOTE2.

Borderline-SMOTE2 not only generates synthetic examples from each example in D and its positive nearest neighbors in P , but also does that from its nearest negative neighbor in N .

— *Borderline-SMOTE: A New Over-Sampling Method in Imbalanced Data Sets Learning*, 2005.

We can implement Borderline-SMOTE1 using the `BorderlineSMOTE` class from `imbalanced-learn`. We can demonstrate the technique on the synthetic binary classification problem used in the previous sections. Instead of generating new synthetic examples for the minority class blindly, we would expect the Borderline-SMOTE method to only create synthetic examples along the decision boundary between the two classes. The complete example of using Borderline-SMOTE to oversample binary classification datasets is listed below.

```
# borderline-SMOTE for imbalanced dataset
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.over_sampling import BorderlineSMOTE
from matplotlib import pyplot
```

```

from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# summarize class distribution
counter = Counter(y)
print(counter)
# transform the dataset
oversample = BorderlineSMOTE()
X, y = oversample.fit_resample(X, y)
# summarize the new class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()

```

Listing 12.20: Example applying Borderline-SMOTE to the imbalanced dataset.

Running the example first creates the dataset and summarizes the initial class distribution, showing a 1:100 relationship. The Borderline-SMOTE is applied to balance the class distribution, which is confirmed with the printed class summary.

```

Counter({0: 9900, 1: 100})
Counter({0: 9900, 1: 9900})

```

Listing 12.21: Example output from applying Borderline-SMOTE to the imbalanced dataset.

Finally, a scatter plot of the transformed dataset is created. The plot clearly shows the effect of the selective approach to oversampling. Examples along the decision boundary of the minority class are oversampled intently (orange). The plot shows that those examples far from the decision boundary are not oversampled. This includes both examples that are easier to classify (those orange points toward the top left of the plot) and those that are overwhelmingly difficult to classify given the strong class overlap (those orange points toward the bottom right of the plot).

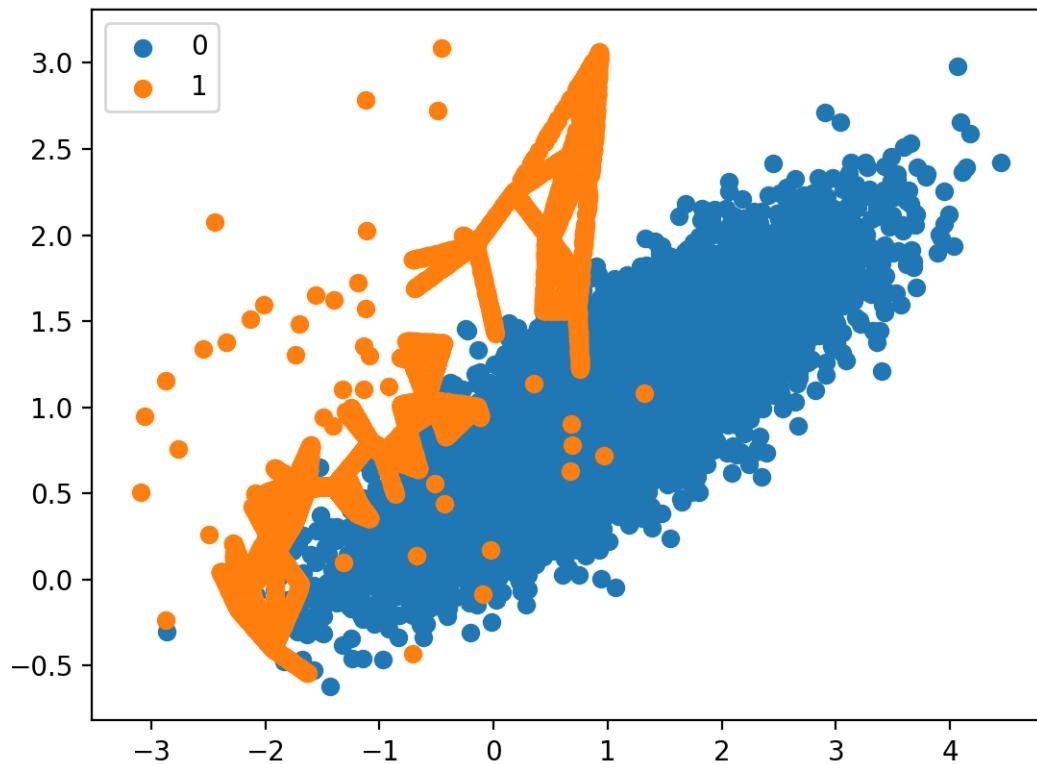


Figure 12.3: Scatter Plot of Imbalanced Dataset With Borderline-SMOTE Oversampling.

12.5.2 Borderline-SMOTE SVM

Hien Nguyen, et al. suggest using an alternative to Borderline-SMOTE where a SVM algorithm is used instead of a KNN to identify misclassified examples on the decision boundary. Their approach is summarized in the 2009 paper titled *Borderline Over-sampling For Imbalanced Data Classification*. A SVM is used to locate the decision boundary defined by the support vectors and examples in the minority class that are close to the support vectors become the focus for generating synthetic examples.

... the borderline area is approximated by the support vectors obtained after training a standard SVMs classifier on the original training set. New instances will be randomly created along the lines joining each minority class support vector with a number of its nearest neighbors using the interpolation

— *Borderline Over-sampling For Imbalanced Data Classification*, 2009.

In addition to using an SVM, the technique attempts to select regions where there are fewer examples of the majority class and tries to extrapolate towards the class boundary.

If majority class instances count for less than a half of its nearest neighbors, new instances will be created with extrapolation to expand minority class area toward the majority class.

— *Borderline Over-sampling For Imbalanced Data Classification*, 2009.

This variation can be implemented via the `SVMSMOTE` class from the imbalanced-learn library. The example below demonstrates this alternative approach to Borderline SMOTE on the same imbalanced dataset.

```
# borderline-SMOTE with SVM for imbalanced dataset
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.over_sampling import SVMSMOTE
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# summarize class distribution
counter = Counter(y)
print(counter)
# transform the dataset
oversample = SVMSMOTE()
X, y = oversample.fit_resample(X, y)
# summarize the new class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Listing 12.22: Example applying SVM-SMOTE to the imbalanced dataset.

Running the example first summarizes the raw class distribution, then the balanced class distribution after applying Borderline-SMOTE with an SVM model.

```
Counter({0: 9900, 1: 100})
Counter({0: 9900, 1: 9900})
```

Listing 12.23: Example output from applying SVM-SMOTE to the imbalanced dataset.

A scatter plot of the dataset is created showing the directed oversampling along the decision boundary with the majority class. We can also see that unlike Borderline-SMOTE, more examples are synthesized away from the region of class overlap, such as toward the top left of the plot.

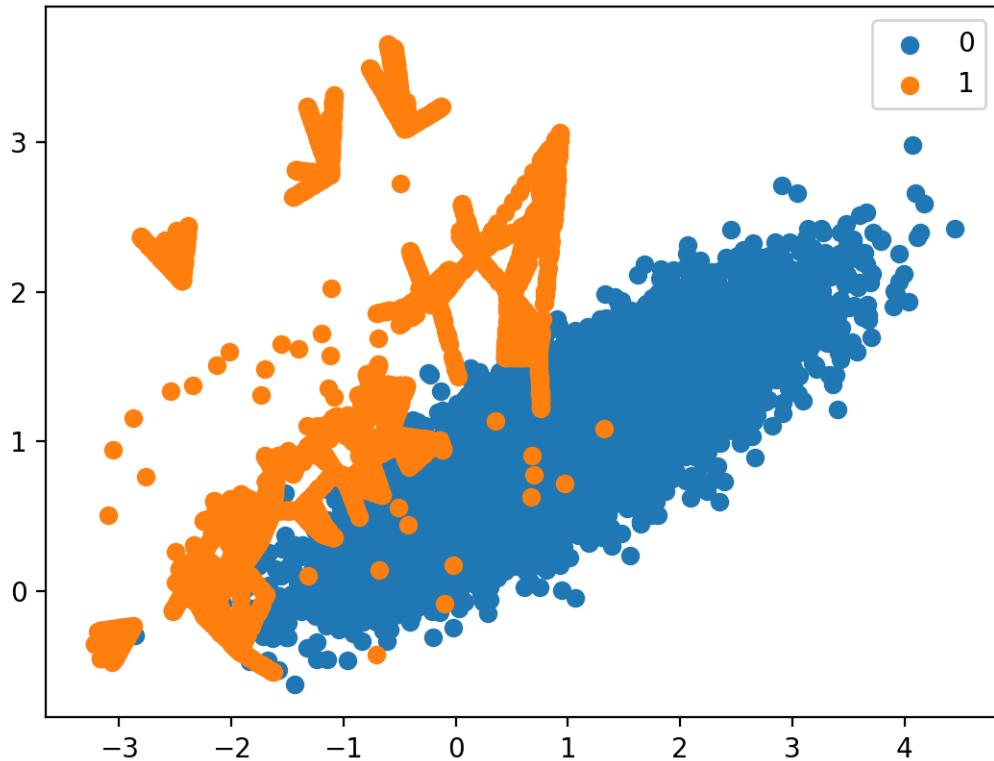


Figure 12.4: Scatter Plot of Imbalanced Dataset With Borderline-SMOTE Oversampling With SVM.

12.5.3 Adaptive Synthetic Sampling (ADASYN)

Another approach involves generating synthetic samples inversely proportional to the density of the examples in the minority class. That is, generate more synthetic examples in regions of the feature space where the density of minority examples is low, and fewer or none where the density is high. This modification to SMOTE is referred to as the Adaptive Synthetic Sampling Method, or ADASYN, and was proposed to Haibo He, et al. in their 2008 paper named for the method titled *ADASYN: Adaptive Synthetic Sampling Approach For Imbalanced Learning*.

ADASYN is based on the idea of adaptively generating minority data samples according to their distributions: more synthetic data is generated for minority class samples that are harder to learn compared to those minority samples that are easier to learn.

— *ADASYN: Adaptive synthetic sampling approach for imbalanced learning*, 2008.

With Borderline-SMOTE, a discriminative model is not created. Instead, examples in the minority class are weighted according to their density, then those examples with the lowest density are the focus for the SMOTE synthetic example generation process.

The key idea of ADASYN algorithm is to use a density distribution as a criterion to automatically decide the number of synthetic samples that need to be generated for each minority data example.

— *ADASYN: Adaptive synthetic sampling approach for imbalanced learning*, 2008.

We can implement this procedure using the ADASYN class in the imbalanced-learn library. The example below demonstrates this alternative approach to oversampling on the imbalanced binary classification dataset.

```
# Oversample and plot imbalanced dataset with ADASYN
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.over_sampling import ADASYN
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# summarize class distribution
counter = Counter(y)
print(counter)
# transform the dataset
oversample = ADASYN()
X, y = oversample.fit_resample(X, y)
# summarize the new class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Listing 12.24: Example applying ADASYN to the imbalanced dataset.

Running the example first creates the dataset and summarizes the initial class distribution, then the updated class distribution after oversampling was performed.

```
Counter({0: 9900, 1: 100})
Counter({0: 9900, 1: 9899})
```

Listing 12.25: Example output from applying ADASYN to the imbalanced dataset.

A scatter plot of the transformed dataset is created. Like Borderline-SMOTE, we can see that synthetic sample generation is focused around the decision boundary as this region has the lowest density. Unlike Borderline-SMOTE, we can see that the examples that have the most class overlap have the most focus. On problems where these low density examples might be outliers, the ADASYN approach may put too much attention on these areas of the feature space, which may result in worse model performance. It may help to remove outliers prior to applying the oversampling procedure, and this might be a helpful heuristic to use more generally.

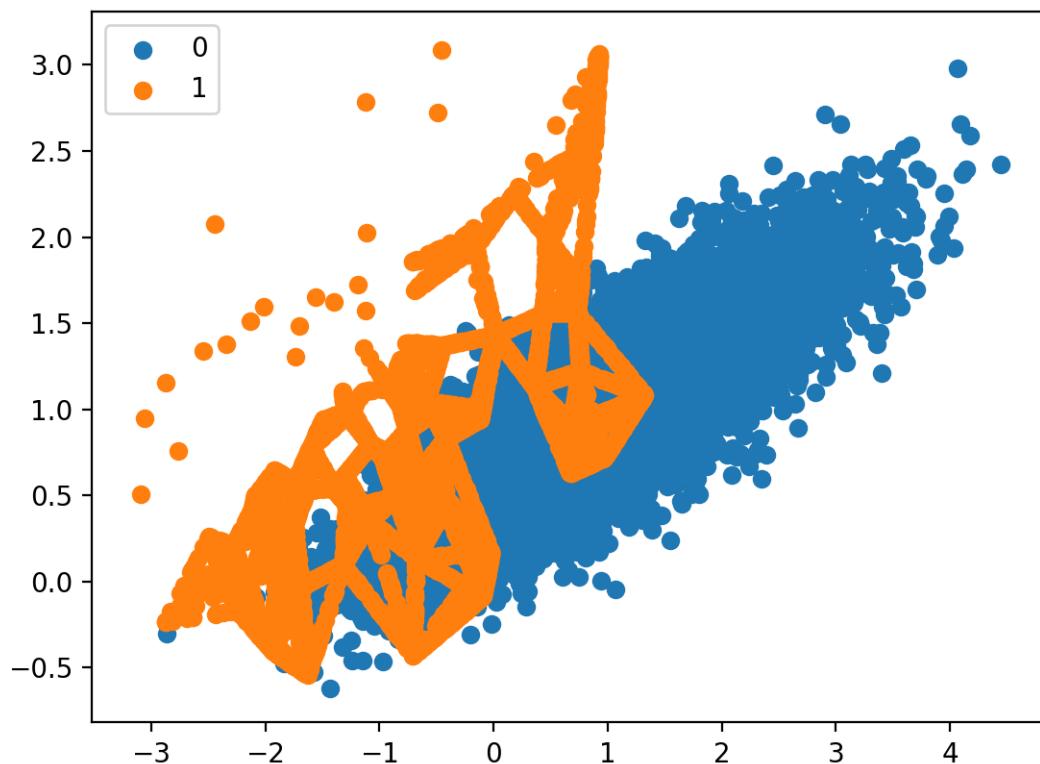


Figure 12.5: Scatter Plot of Imbalanced Dataset With Adaptive Synthetic Sampling (ADASYN).

12.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

12.6.1 Papers

- *SMOTE: Synthetic Minority Over-sampling Technique*, 2002.
<https://arxiv.org/abs/1106.1813>
- *Borderline-SMOTE: A New Over-Sampling Method in Imbalanced Data Sets Learning*, 2005.
https://link.springer.com/chapter/10.1007/11538059_91
- *Borderline Over-sampling For Imbalanced Data Classification*, 2009.
<http://ousar.lib.okayama-u.ac.jp/en/19617>
- *ADASYN: Adaptive Synthetic Sampling Approach For Imbalanced Learning*, 2008.
<https://ieeexplore.ieee.org/document/4633969>

12.6.2 Books

- *Learning from Imbalanced Data Sets*, 2018.
<https://amzn.to/307Xlva>
- *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.
<https://amzn.to/32K9K6d>

12.6.3 API

- `imblearn.over_sampling.SMOTE` API.
https://imbalanced-learn.org/stable/generated/imblearn.over_sampling.SMOTE.html
- `imblearn.over_sampling.SMOTENC` API.
https://imbalanced-learn.org/stable/generated/imblearn.over_sampling.SMOTENC.html
- `imblearn.over_sampling.BorderlineSMOTE` API.
https://imbalanced-learn.org/stable/generated/imblearn.over_sampling.BorderlineSMOTE.html
- `imblearn.over_sampling.SVMSMOTE` API.
https://imbalanced-learn.org/stable/generated/imblearn.over_sampling.SVMSMOTE.html
- `imblearn.over_sampling.ADASYN` API.
https://imbalanced-learn.org/stable/generated/imblearn.over_sampling.ADASYN.html

12.6.4 Articles

- Oversampling and undersampling in data analysis, Wikipedia.
https://en.wikipedia.org/wiki/Oversampling_and_undersampling_in_data_analysis

12.7 Summary

In this tutorial, you discovered the SMOTE for oversampling imbalanced classification datasets. Specifically, you learned:

- How the SMOTE synthesizes new examples for the minority class.
- How to correctly fit and evaluate machine learning models on SMOTE-transformed training datasets.
- How to use extensions of the SMOTE that generate synthetic examples along the class decision boundary.

12.7.1 Next

In the next tutorial, you will discover a suite of undersampling techniques to change the class distribution of the training dataset.

Chapter 13

Undersampling Methods

Resampling methods are designed to change the composition of a training dataset for an imbalanced classification task. Most of the attention of sampling methods for imbalanced classification is put on oversampling the minority class. Nevertheless, a suite of techniques has been developed for undersampling the majority class that can be used in conjunction with effective oversampling methods.

There are many different types of undersampling techniques, although most can be grouped into those that select examples to keep in the transformed dataset, those that select examples to delete, and hybrids that combine both types of methods. In this tutorial, you will discover undersampling methods for imbalanced classification. After completing this tutorial, you will know:

- How to use the Near-Miss and Condensed Nearest Neighbor Rule methods that select examples to keep in the majority class.
- How to use Tomek Links and the Edited Nearest Neighbors Rule methods that select examples to delete from the majority class.
- How to use One-Sided Selection and the Neighborhood Cleaning Rule that combine methods for choosing examples to keep and delete from the majority class.

Let's get started.

Note: This chapter makes use of the imbalanced-learn library. See Appendix B for installation instructions, if needed.

13.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Undersampling for Imbalanced Classification
2. Methods that Select Examples to Keep
3. Methods that Select Examples to Delete
4. Combinations of Keep and Delete Methods

13.2 Undersampling for Imbalanced Classification

Undersampling refers to a group of techniques designed to balance the class distribution for a classification dataset that has a skewed class distribution. An imbalanced class distribution will have one or more classes with relatively few examples (the minority classes) and one or more classes with relatively many examples (the majority classes). It is best understood in the context of a binary (two-class) classification problem where class 0 is the majority class and class 1 is the minority class.

Undersampling techniques remove examples from the training dataset that belong to the majority class in order to better balance the class distribution, such as reducing the skew from a 1:100 to a 1:10, 1:2, or even a 1:1 class distribution. This is different from oversampling that involves adding examples to the minority class in an effort to reduce the skew in the class distribution.

... undersampling, that consists of reducing the data by eliminating examples belonging to the majority class with the objective of equalizing the number of examples of each class ...

— Page 82, *Learning from Imbalanced Data Sets*, 2018.

Undersampling methods can be used directly on a training dataset that can then, in turn, be used to fit a machine learning model. Typically, undersampling methods are used in conjunction with an oversampling technique for the minority class, and this combination often results in better performance than using oversampling or undersampling alone on the training dataset.

The simplest undersampling technique involves randomly selecting examples from the majority class and deleting them from the training dataset. This is referred to as random undersampling. Although simple and effective, a limitation of this technique is that examples are removed without any concern for how useful or important they might be in determining the decision boundary between the classes. This means it is possible, or even likely, that useful information will be deleted.

The major drawback of random undersampling is that this method can discard potentially useful data that could be important for the induction process. The removal of data is a critical decision to be made, hence many of the proposed undersampling [methods] use heuristics in order to overcome the limitations of the non-heuristics decisions.

— Page 83, *Learning from Imbalanced Data Sets*, 2018.

An extension of this approach is to be more discerning regarding the examples from the majority class that are deleted. This typically involves heuristics or learning models that attempt to identify redundant examples for deletion or useful examples for non-deletion. There are many undersampling techniques that use these types of heuristics. In the following sections, we will review some of the more common methods and develop an intuition for their operation on a synthetic imbalanced binary classification dataset.

We can define a synthetic binary classification dataset using the `make_classification()` function from the scikit-learn library. For example, we can create 10,000 examples with two input variables and a 1:100 distribution as follows:

```
...
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
```

Listing 13.1: Example of defining an imbalanced binary classification problem.

We can then create a scatter plot of the dataset via the `scatter()` Matplotlib function to understand the spatial relationship of the examples in each class and their imbalance.

```
...
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Listing 13.2: Example of creating a scatter plot with dots colored by class label.

Tying this together, the complete example of creating an imbalanced classification dataset and plotting the examples is listed below.

```
# Generate and plot a synthetic imbalanced classification dataset
from collections import Counter
from sklearn.datasets import make_classification
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# summarize class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Listing 13.3: Example of defining and summarizing an imbalanced classification dataset.

Running the example first summarizes the class distribution, showing an approximate 1:100 class distribution with about 10,000 examples with class 0 and 100 with class 1.

```
Counter({0: 9900, 1: 100})
```

Listing 13.4: Example output from defining and summarizing an imbalanced classification dataset.

Next, a scatter plot is created showing all of the examples in the dataset. We can see a large mass of examples for class 0 (blue) and a small number of examples for class 1 (orange). We can also see that the classes overlap with some examples from class 1 clearly within the part of the feature space that belongs to class 0.

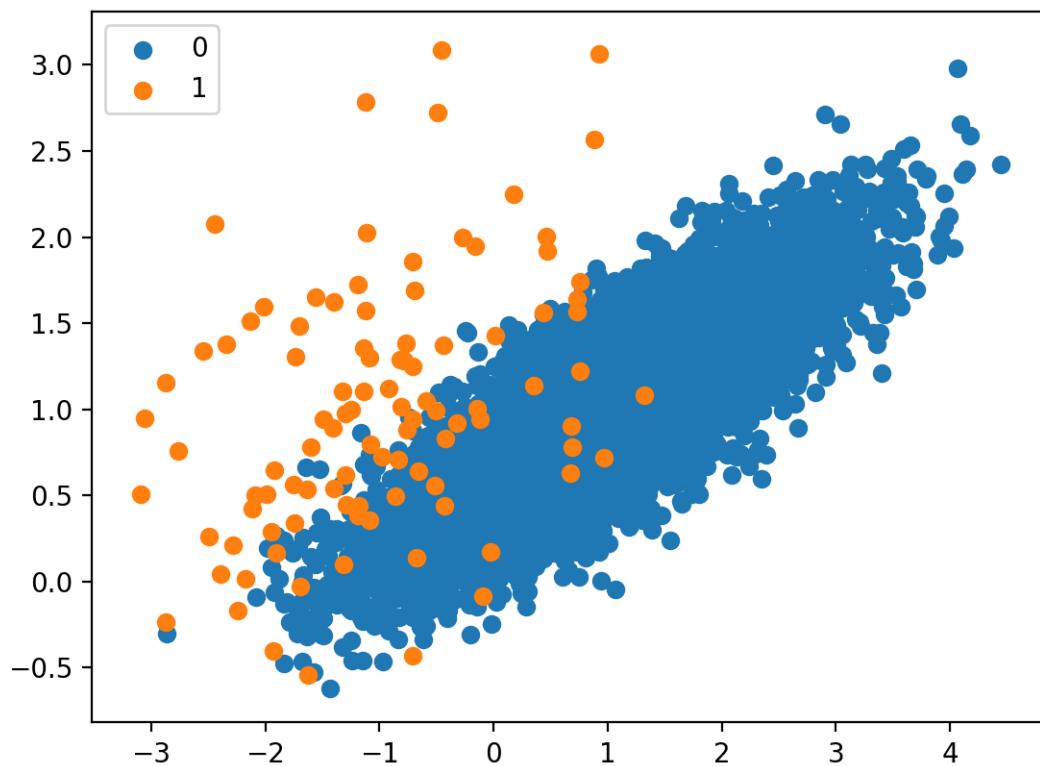


Figure 13.1: Scatter Plot of Imbalanced Classification Dataset.

This plot provides the starting point for developing the intuition for the effect that different undersampling techniques have on the majority class. Next, we can begin to review popular undersampling methods made available via the imbalanced-learn Python library. There are many different methods to choose from. We will divide them into methods that select examples from the majority class to keep, methods that select examples to delete, and combinations of both approaches.

13.3 Methods that Select Examples to Keep

In this section, we will take a closer look at two methods that choose which examples from the majority class to keep, the near-miss family of methods, and the popular condensed nearest neighbor rule.

13.3.1 Near Miss Undersampling

Near Miss refers to a collection of undersampling methods that select examples based on the distance of majority class examples to minority class examples. The approaches were proposed by Jianping Zhang and Inderjeet Mani in their 2003 paper titled *KNN Approach to Unbalanced*

Data Distributions: A Case Study Involving Information Extraction. There are three versions of the technique, named NearMiss-1, NearMiss-2, and NearMiss-3.

NearMiss-1 selects examples from the majority class that have the smallest average distance to the three closest examples from the minority class. NearMiss-2 selects examples from the majority class that have the smallest average distance to the three furthest examples from the minority class. NearMiss-3 involves selecting a given number of majority class examples for each example in the minority class that are closest. Here, distance is determined in feature space using Euclidean distance or similar.

- **NearMiss-1:** Majority class examples with minimum average distance to three closest minority class examples.
- **NearMiss-2:** Majority class examples with minimum average distance to three furthest minority class examples.
- **NearMiss-3:** Majority class examples with minimum distance to each minority class example.

The NearMiss-3 seems desirable, given that it will only keep those majority class examples that are on the decision boundary. We can implement the Near Miss methods using the NearMiss imbalanced-learn class. The type of near-miss strategy used is defined by the `version` argument, which by default is set to 1 for NearMiss-1, but can be set to 2 or 3 for the other two methods.

```
...
# define the undersampling method
undersample = NearMiss(version=1)
```

Listing 13.5: Example of defining the NearMiss undersampling strategy.

By default, the technique will undersample the majority class to have the same number of examples as the minority class, although this can be changed by setting the `sampling_strategy` argument to a fraction of the minority class. First, we can demonstrate NearMiss-1 that selects only those majority class examples that have a minimum distance to three minority class instances, defined by the `n_neighbors` argument. We would expect clusters of majority class examples around the minority class examples that overlap. The complete example is listed below.

```
# Undersample imbalanced dataset with NearMiss-1
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.under_sampling import NearMiss
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# summarize class distribution
counter = Counter(y)
print(counter)
# define the undersampling method
undersample = NearMiss(version=1, n_neighbors=3)
# transform the dataset
X, y = undersample.fit_resample(X, y)
```

```
# summarize the new class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Listing 13.6: Example of applying the NearMiss-1 strategy to the imbalanced classification dataset.

Running the example undersamples the majority class and creates a scatter plot of the transformed dataset. We can see that, as expected, only those examples in the majority class that are closest to the minority class examples in the overlapping area were retained.

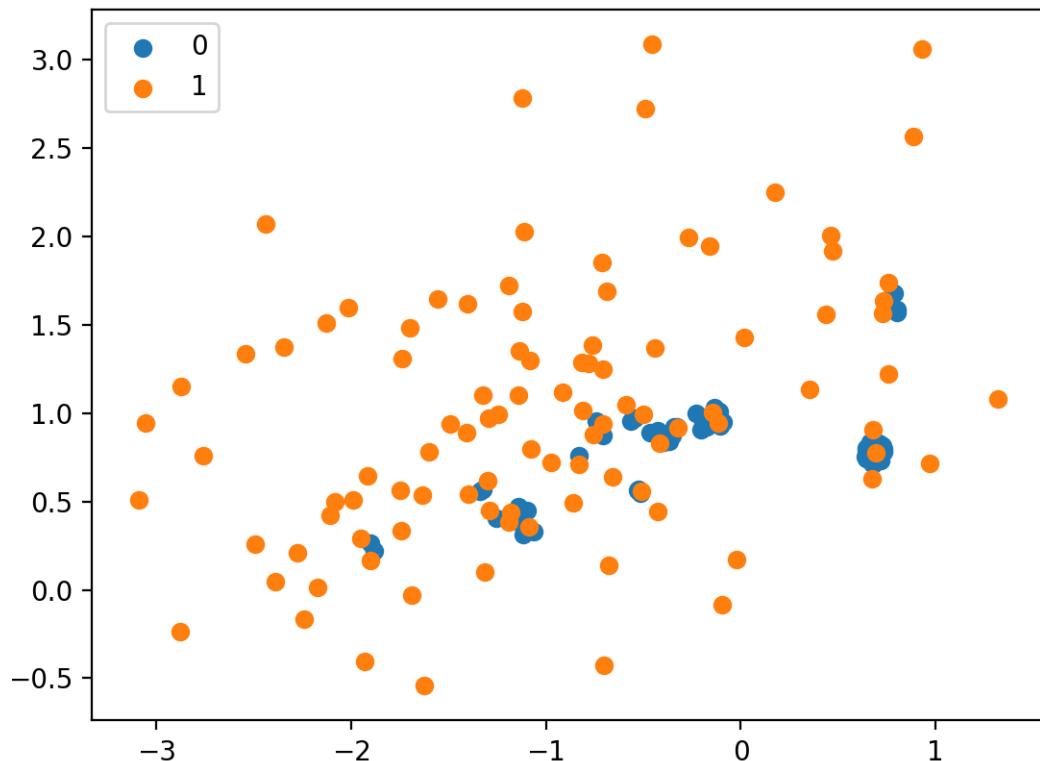


Figure 13.2: Scatter Plot of Imbalanced Dataset Undersampled with NearMiss-1.

Next, we can demonstrate the NearMiss-2 strategy, which is an inverse to NearMiss-1. It selects examples that are closest to the most distant examples from the minority class, defined by the `n_neighbors` argument. This is not an intuitive strategy from the description alone. The complete example is listed below.

```
# Undersample imbalanced dataset with NearMiss-2
```

```
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.under_sampling import NearMiss
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# summarize class distribution
counter = Counter(y)
print(counter)
# define the undersampling method
undersample = NearMiss(version=2, n_neighbors=3)
# transform the dataset
X, y = undersample.fit_resample(X, y)
# summarize the new class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Listing 13.7: Example of applying the NearMiss-2 strategy to the imbalanced classification dataset.

Running the example, we can see that the NearMiss-2 selects examples that appear to be in the center of mass for the overlap between the two classes.

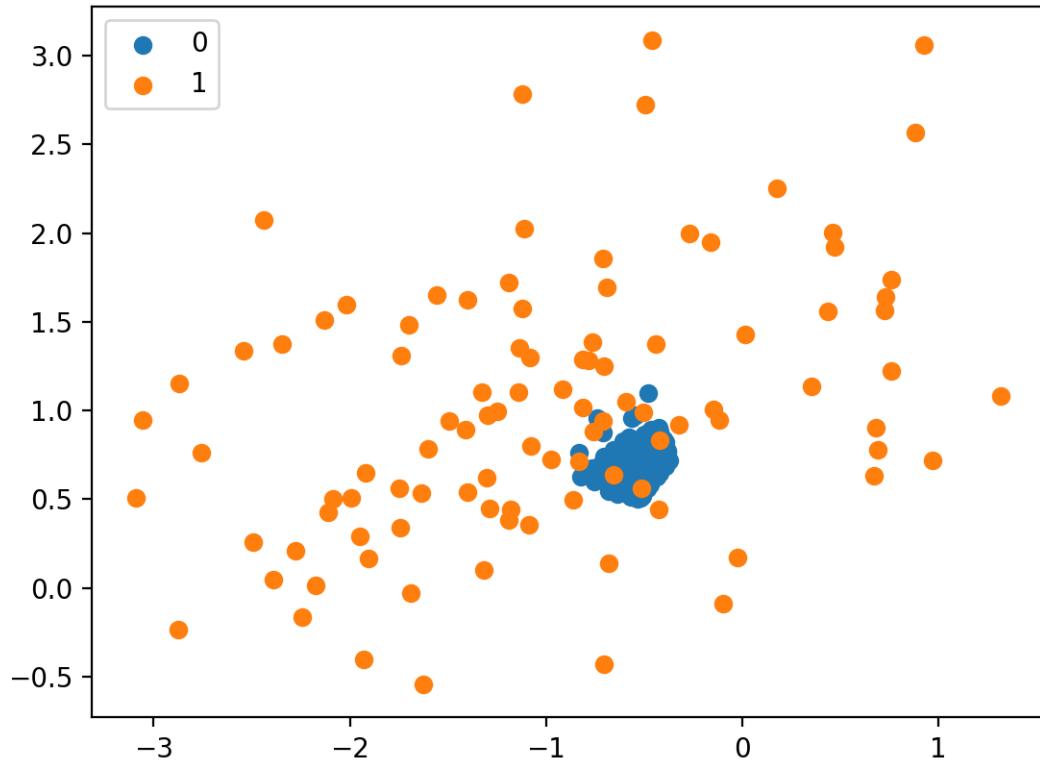


Figure 13.3: Scatter Plot of Imbalanced Dataset Undersampled With NearMiss-2.

Finally, we can try NearMiss-3 that selects the closest examples from the majority class for each minority class. The `n_neighbors_ver3` argument determines the number of examples to select for each minority example, although the desired balancing ratio set via `sampling_strategy` will filter this so that the desired balance is achieved. The complete example is listed below.

```
# Undersample imbalanced dataset with NearMiss-3
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.under_sampling import NearMiss
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# summarize class distribution
counter = Counter(y)
print(counter)
# define the undersampling method
undersample = NearMiss(version=3, n_neighbors_ver3=3)
# transform the dataset
X, y = undersample.fit_resample(X, y)
# summarize the new class distribution
counter = Counter(y)
```

```

print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()

```

Listing 13.8: Example of applying the NearMiss-3 strategy to the imbalanced classification dataset.

As expected, we can see that each example in the minority class that was in the region of overlap with the majority class has up to three neighbors from the majority class.

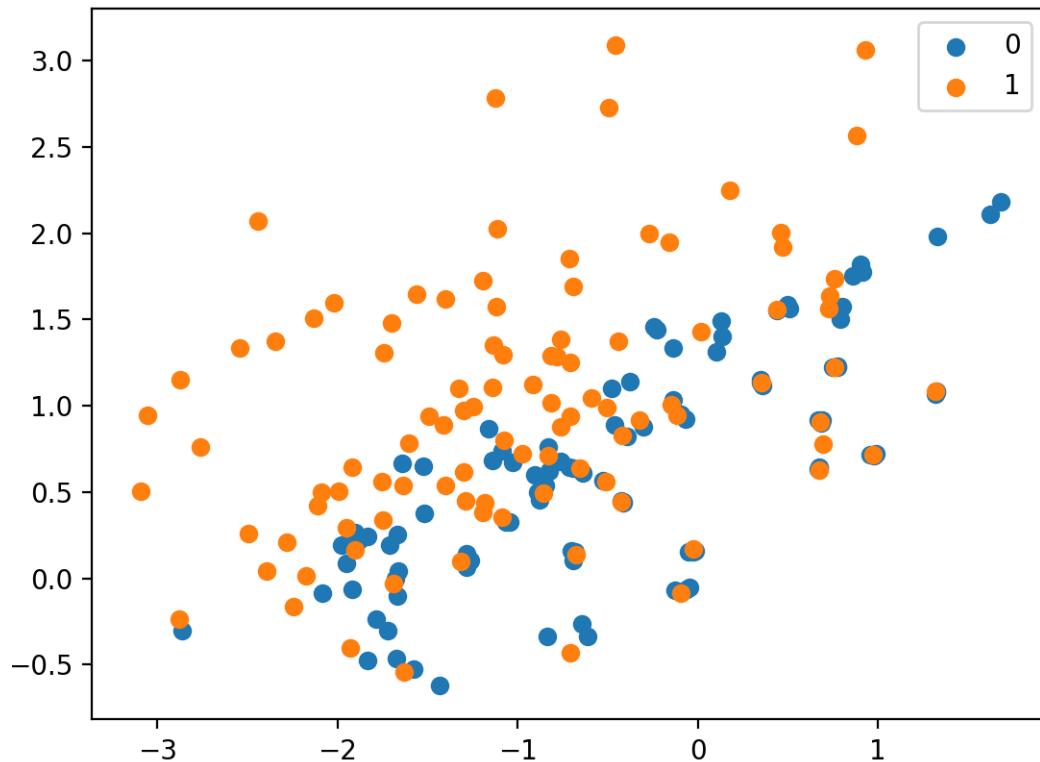


Figure 13.4: Scatter Plot of Imbalanced Dataset Undersampled With NearMiss-3.

13.3.2 Condensed Nearest Neighbor Rule Undersampling

Condensed Nearest Neighbors, or CNN for short, is an undersampling technique that seeks a subset of a collection of samples that results in no loss in model performance, referred to as a minimal consistent set.

... the notion of a consistent subset of a sample set. This is a subset which, when used as a stored reference set for the NN rule, correctly classifies all of the remaining points in the sample set.

— *The Condensed Nearest Neighbor Rule (Corresp.), 1968.*

It is achieved by enumerating the examples in the dataset and adding them to the *store* only if they cannot be classified correctly by the current contents of the store. This approach was proposed to reduce the memory requirements for the *k*-Nearest Neighbors (KNN) algorithm by Peter Hart in the 1968 correspondence titled *The Condensed Nearest Neighbor Rule*.

When used for imbalanced classification, the store is comprised of all examples in the minority set and only examples from the majority set that cannot be classified correctly are added incrementally to the store. We can implement the Condensed Nearest Neighbor for undersampling using the `CondensedNearestNeighbour` class from the imbalanced-learn library. During the procedure, the KNN algorithm is used to classify points to determine if they are to be added to the store or not. The *k* value is set via the `n_neighbors` argument and defaults to 1.

```
...
# define the undersampling method
undersample = CondensedNearestNeighbour(n_neighbors=1)
```

Listing 13.9: Example of defining the CondensedNearestNeighbour undersampling strategy.

It's a relatively slow procedure, so small datasets and small *k* values are preferred. The complete example of demonstrating the Condensed Nearest Neighbor rule for undersampling is listed below.

```
# undersample and plot imbalanced dataset with the Condensed Nearest Neighbor Rule
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.under_sampling import CondensedNearestNeighbour
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# summarize class distribution
counter = Counter(y)
print(counter)
# define the undersampling method
undersample = CondensedNearestNeighbour(n_neighbors=1)
# transform the dataset
X, y = undersample.fit_resample(X, y)
# summarize the new class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Listing 13.10: Example of applying the CNN strategy to the imbalanced classification dataset.

Running the example first reports the skewed distribution of the raw dataset, then the more balanced distribution for the transformed dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the resulting distribution is about 1:2 minority to majority examples. This highlights that although the `sampling_strategy` argument seeks to balance the class distribution, the algorithm will continue to add misclassified examples to the store (transformed dataset). This is a desirable property.

```
Counter({0: 9900, 1: 100})
Counter({0: 188, 1: 100})
```

Listing 13.11: Example output from applying the CNN strategy to the imbalanced classification dataset.

A scatter plot of the resulting dataset is created. We can see that the focus of the algorithm is those examples in the minority class along the decision boundary between the two classes, specifically, those majority examples around the minority class examples.

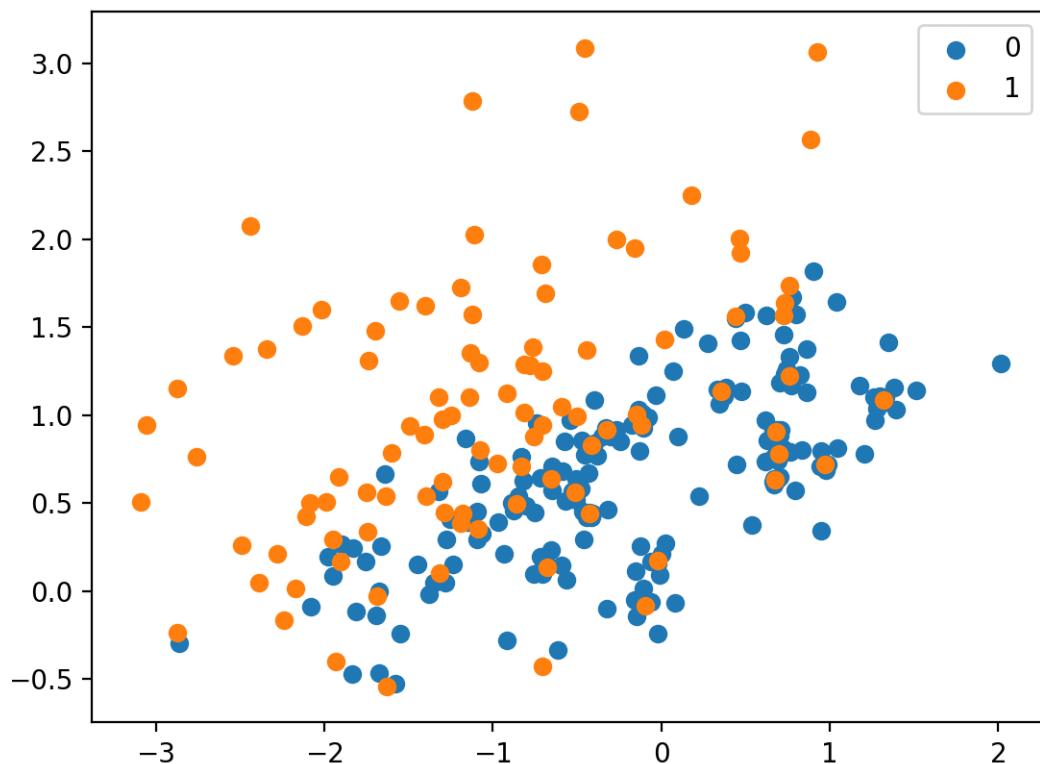


Figure 13.5: Scatter Plot of Imbalanced Dataset Undersampled With the Condensed Nearest Neighbor Rule.

13.4 Methods that Select Examples to Delete

In this section, we will take a closer look at methods that select examples from the majority class to delete, including the popular Tomek Links method and the Edited Nearest Neighbors rule.

13.4.1 Tomek Links for Undersampling

A criticism of the Condensed Nearest Neighbor Rule is that examples are selected randomly, especially initially. This has the effect of allowing redundant examples into the store and in allowing examples that are internal to the mass of the distribution, rather than on the class boundary, into the store.

The condensed nearest-neighbor (CNN) method chooses samples randomly. This results in a) retention of unnecessary samples and b) occasional retention of internal rather than boundary samples.

— *Two modifications of CNN*, 1976.

Two modifications to the CNN procedure were proposed by Ivan Tomek in his 1976 paper titled *Two modifications of CNN*. One of the modifications (Method2) is a rule that finds pairs of examples, one from each class; they together have the smallest Euclidean distance to each other in feature space. This means that in a binary classification problem with classes 0 and 1, a pair would have an example from each class and would be closest neighbors across the dataset.

In words, instances a and b define a Tomek Link if: (i) instance a 's nearest neighbor is b , (ii) instance b 's nearest neighbor is a , and (iii) instances a and b belong to different classes.

— Page 46, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

These cross-class pairs are now generally referred to as *Tomek Links* and are valuable as they define the class boundary.

Method 2 has another potentially important property: It finds pairs of boundary points which participate in the formation of the (piecewise-linear) boundary. [...] Such methods could use these pairs to generate progressively simpler descriptions of acceptably accurate approximations of the original completely specified boundaries.

— *Two modifications of CNN*, 1976.

The procedure for finding Tomek Links can be used to locate all cross-class nearest neighbors. If the examples in the minority class are held constant, the procedure can be used to find all of those examples in the majority class that are closest to the minority class, then removed. These would be the ambiguous examples.

From this definition, we see that instances that are in Tomek Links are either boundary instances or noisy instances. This is due to the fact that only boundary instances and noisy instances will have nearest neighbors, which are from the opposite class.

— Page 46, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

We can implement Tomek Links method for undersampling using the `TomekLinks` imbalanced-learn class.

```
...
# define the undersampling method
undersample = TomekLinks()
```

Listing 13.12: Example of defining the TomekLinks undersampling strategy.

The complete example of demonstrating the Tomek Links for undersampling is listed below. Because the procedure only removes so-named *Tomek Links*, we would not expect the resulting transformed dataset to be balanced, only less ambiguous along the class boundary.

```
# undersample and plot imbalanced dataset with Tomek Links
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.under_sampling import TomekLinks
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# summarize class distribution
counter = Counter(y)
print(counter)
# define the undersampling method
undersample = TomekLinks()
# transform the dataset
X, y = undersample.fit_resample(X, y)
# summarize the new class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Listing 13.13: Example of applying the Tomek Links strategy to the imbalanced classification dataset.

Running the example first summarizes the class distribution for the raw dataset, then the transformed dataset. We can see that only 26 examples from the majority class were removed.

```
Counter({0: 9900, 1: 100})
Counter({0: 9874, 1: 100})
```

Listing 13.14: Example output from applying the Tomek Links strategy to the imbalanced classification dataset.

The scatter plot of the transformed dataset does not make the minor editing to the majority class obvious. This highlights that although finding the ambiguous examples on the class boundary is useful, it is not a great undersampling technique on its own. In practice, the Tomek

Links procedure is often combined with other methods, such as the Condensed Nearest Neighbor Rule.

The choice to combine Tomek Links and CNN is natural, as Tomek Links can be said to remove borderline and noisy instances, while CNN removes redundant instances.

— Page 46, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

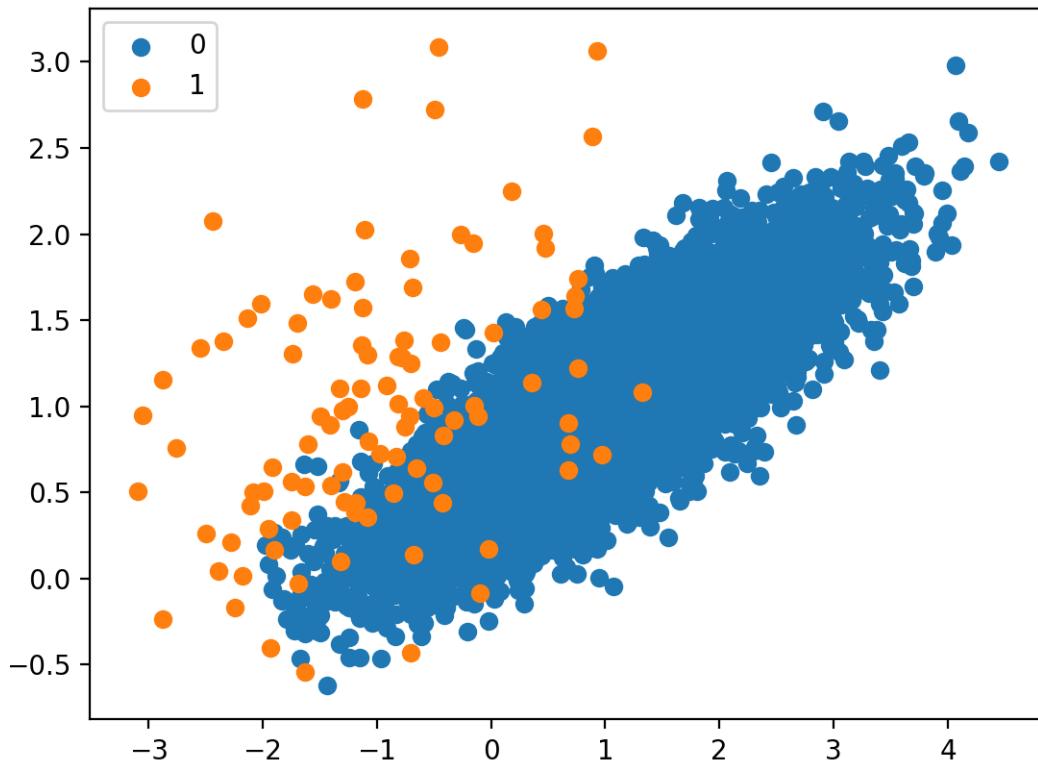


Figure 13.6: Scatter Plot of Imbalanced Dataset Undersampled With the Tomek Links Method.

13.4.2 Edited Nearest Neighbors Rule for Undersampling

Another rule for finding ambiguous and noisy examples in a dataset is called Edited Nearest Neighbors, or sometimes ENN for short. This rule involves using $k = 3$ nearest neighbors to locate those examples in a dataset that are misclassified and that are then removed before a $k = 1$ classification rule is applied. This approach of sampling and classification was proposed by Dennis Wilson in his 1972 paper titled *Asymptotic Properties of Nearest Neighbor Rules Using Edited Data*.

The modified three-nearest neighbor rule which uses the three-nearest neighbor rule to edit the preclassified samples and then uses a single-nearest neighbor rule to make decisions is a particularly attractive rule.

— *Asymptotic Properties of Nearest Neighbor Rules Using Edited Data*, 1972.

When used as an undersampling procedure, the rule can be applied to each example in the majority class, allowing those examples that are misclassified as belonging to the minority class to be removed, and those correctly classified to remain. It is also applied to each example in the minority class where those examples that are misclassified have their nearest neighbors from the majority class deleted.

... for each instance a in the dataset, its three nearest neighbors are computed. If a is a majority class instance and is misclassified by its three nearest neighbors, then a is removed from the dataset. Alternatively, if a is a minority class instance and is misclassified by its three nearest neighbors, then the majority class instances among a 's neighbors are removed.

— Page 46, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

The Edited Nearest Neighbors rule can be implemented using the `EditedNearestNeighbours` imbalanced-learn class. The `n_neighbors` argument controls the number of neighbors to use in the editing rule, which defaults to three, as in the paper.

```
...
# define the undersampling method
undersample = EditedNearestNeighbours(n_neighbors=3)
```

Listing 13.15: Example of defining the ENN undersampling strategy.

The complete example of demonstrating the ENN rule for undersampling is listed below. Like Tomek Links, the procedure only removes noisy and ambiguous points along the class boundary. As such, we would not expect the resulting transformed dataset to be balanced.

```
# undersample and plot imbalanced dataset with the Edited Nearest Neighbor rule
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.under_sampling import EditedNearestNeighbours
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# summarize class distribution
counter = Counter(y)
print(counter)
# define the undersampling method
undersample = EditedNearestNeighbours(n_neighbors=3)
# transform the dataset
X, y = undersample.fit_resample(X, y)
# summarize the new class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
```

```
pyplot.show()
```

Listing 13.16: Example of applying the ENN strategy to the imbalanced classification dataset.

Running the example first summarizes the class distribution for the raw dataset, then the transformed dataset. We can see that only 94 examples from the majority class were removed.

```
Counter({0: 9900, 1: 100})
Counter({0: 9806, 1: 100})
```

Listing 13.17: Example output from applying the ENN strategy to the imbalanced classification dataset.

Given the small amount of undersampling performed, the change to the mass of majority examples is not obvious from the plot. Also, like Tomek Links, the Edited Nearest Neighbor Rule gives best results when combined with another undersampling method.

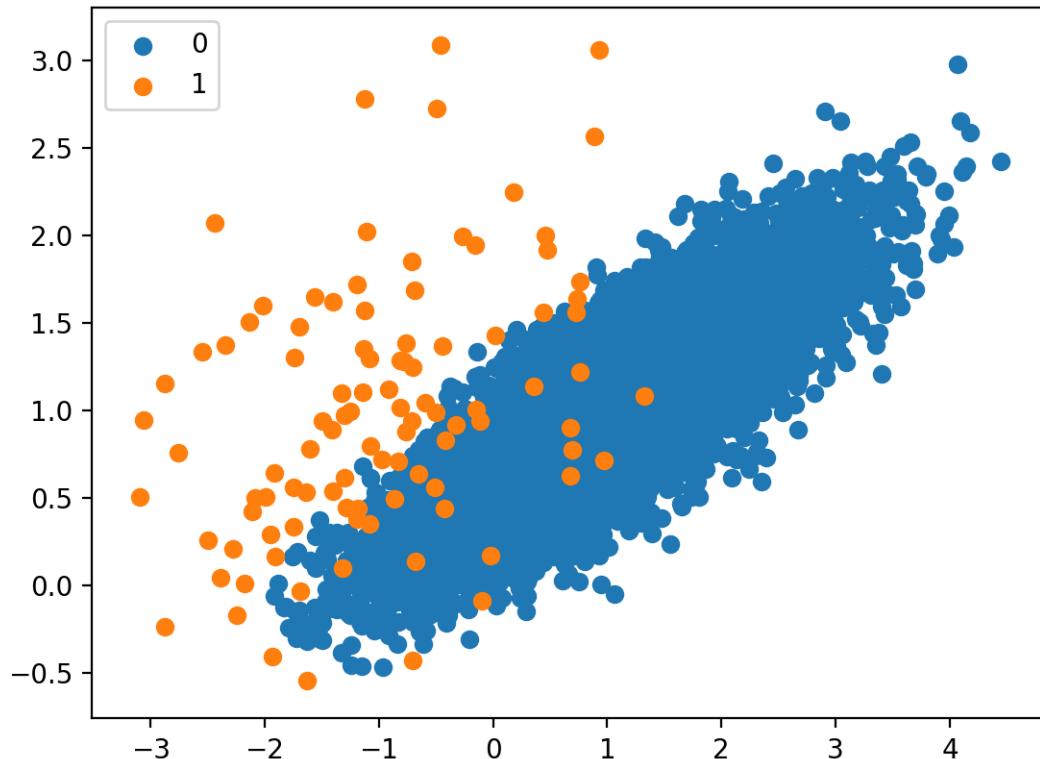


Figure 13.7: Scatter Plot of Imbalanced Dataset Undersampled With the Edited Nearest Neighbor Rule.

Ivan Tomek, developer of Tomek Links, explored extensions of the Edited Nearest Neighbor Rule in his 1976 paper titled *An Experiment with the Edited Nearest-Neighbor Rule*. Among his experiments was a repeated ENN method that invoked the continued editing of the dataset using the ENN rule for a fixed number of iterations, referred to as *unlimited editing*.

... unlimited repetition of Wilson's editing (in fact, editing is always stopped after a finite number of steps because after a certain number of repetitions the design set becomes immune to further elimination)

— *An Experiment with the Edited Nearest-Neighbor Rule*, 1976.

He also describes a method referred to as *All KNN* that removes all examples from the dataset that were classified incorrectly. Both of these additional editing procedures are also available via the imbalanced-learn library via the `RepeatedEditedNearestNeighbours` and `AllKNN` classes.

13.5 Combinations of Keep and Delete Methods

In this section, we will take a closer look at techniques that combine the techniques we have already looked at to both keep and delete examples from the majority class, such as One-Sided Selection and the Neighborhood Cleaning Rule.

13.5.1 One-Sided Selection for Undersampling

One-Sided Selection, or OSS for short, is an undersampling technique that combines Tomek Links and the Condensed Nearest Neighbor (CNN) Rule. Specifically, Tomek Links are ambiguous points on the class boundary and are identified and removed in the majority class. The CNN method is then used to remove redundant examples from the majority class that are far from the decision boundary.

OSS is an undersampling method resulting from the application of Tomek links followed by the application of US-CNN. Tomek links are used as an undersampling method and removes noisy and borderline majority class examples. [...] US-CNN aims to remove examples from the majority class that are distant from the decision border.

— Page 84, *Learning from Imbalanced Data Sets*, 2018.

This combination of methods was proposed by Miroslav Kubat and Stan Matwin in their 1997 paper titled *Addressing The Curse Of Imbalanced Training Sets: One-sided Selection*. The CNN procedure occurs in one-step and involves first adding all minority class examples to the store and some number of majority class examples (e.g. 1), then classifying all remaining majority class examples with KNN ($k = 1$) and adding those that are misclassified to the store.

1. Let S be the original training set.
2. Initially, C contains all positive examples from S and one randomly selected negative example.
3. Classify S with the 1-NN rule using the examples in C , and compare the assigned concept labels with the original ones. Move all misclassified examples into C that is now consistent with S while being smaller.
4. Remove from C all negative examples participating in Tomek links. This removes those negative examples that are believed borderline and/or noisy. All positive examples are retained. The resulting set is referred to as T .

Figure 13.8: Overview of the One-Sided Selection for Undersampling Procedure. Taken from *Addressing The Curse Of Imbalanced Training Sets: One-sided Selection*.

We can implement the OSS undersampling strategy via the `OneSidedSelection` imbalanced-learn class. The number of seed examples can be set with `n_seeds_S` and defaults to 1 and the k for KNN can be set via the `n_neighbors` argument and defaults to 1. Given that the CNN procedure occurs in one block, it is more useful to have a larger seed sample of the majority class in order to effectively remove redundant examples. In this case, we will use a value of 200.

```
...
# define the undersampling method
undersample = OneSidedSelection(n_neighbors=1, n_seeds_S=200)
```

Listing 13.18: Example of defining the OSS undersampling strategy.

The complete example of applying OSS on the binary classification problem is listed below. We might expect a large number of redundant examples from the majority class to be removed from the interior of the distribution (e.g. away from the class boundary).

```
# undersample and plot imbalanced dataset with One-Sided Selection
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.under_sampling import OneSidedSelection
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# summarize class distribution
counter = Counter(y)
print(counter)
# define the undersampling method
undersample = OneSidedSelection(n_neighbors=1, n_seeds_S=200)
# transform the dataset
X, y = undersample.fit_resample(X, y)
# summarize the new class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
```

```

row_ix = where(y == label)[0]
pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()

```

Listing 13.19: Example of applying the OSS strategy to the imbalanced classification dataset.

Running the example first reports the class distribution in the raw dataset, then the transformed dataset. We can see that a large number of examples from the majority class were removed, consisting of both redundant examples (removed via CNN) and ambiguous examples (removed via Tomek Links). The ratio for this dataset is now around 1:10, down from 1:100.

```

Counter({0: 9900, 1: 100})
Counter({0: 940, 1: 100})

```

Listing 13.20: Example output from applying the OSS strategy to the imbalanced classification dataset.

A scatter plot of the transformed dataset is created showing that most of the majority class examples left are around the class boundary and the overlapping examples from the minority class. It might be interesting to explore larger seed samples from the majority class and different values of k used in the one-step CNN procedure.

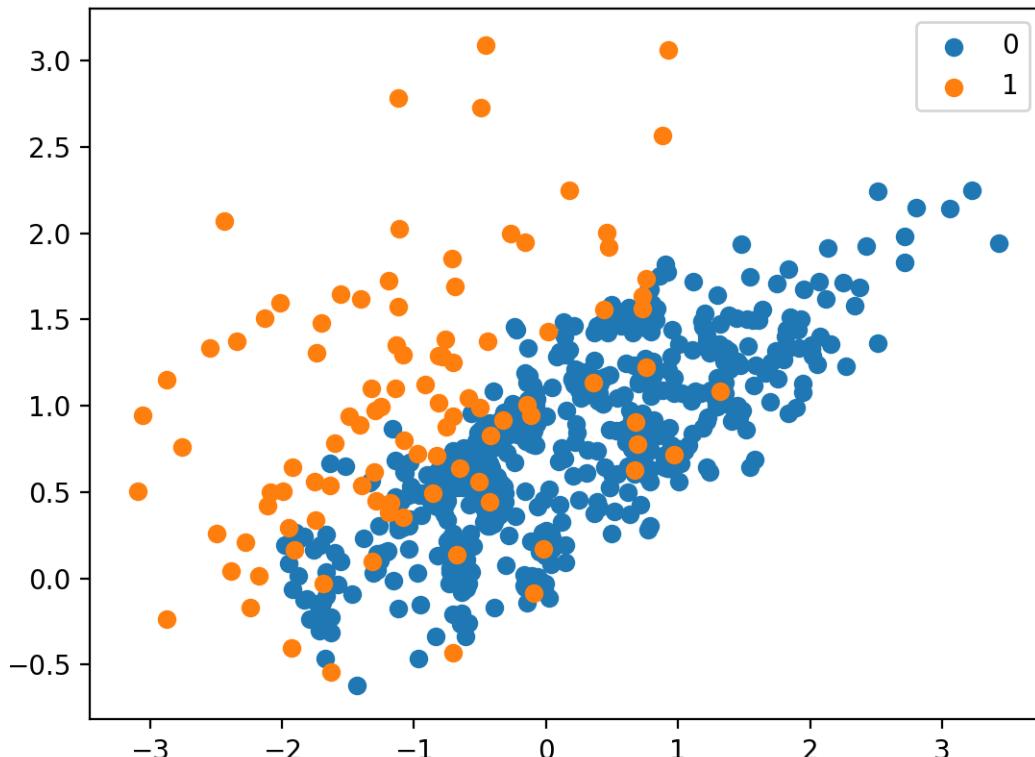


Figure 13.9: Scatter Plot of Imbalanced Dataset Undersampled With One-Sided Selection.

13.5.2 Neighborhood Cleaning Rule for Undersampling

The Neighborhood Cleaning Rule, or NCR for short, is an undersampling technique that combines both the Condensed Nearest Neighbor (CNN) Rule to remove redundant examples and the Edited Nearest Neighbors (ENN) Rule to remove noisy or ambiguous examples.

Like One-Sided Selection (OSS), the CNN method is applied in a one-step manner, then the examples that are misclassified according to a KNN classifier are removed, as per the ENN rule. Unlike OSS, less of the redundant examples are removed and more attention is placed on *cleaning* those examples that are retained. The reason for this is to focus less on improving the balance of the class distribution and more on the quality (unambiguity) of the examples that are retained in the majority class.

... the quality of classification results does not necessarily depend on the size of the class. Therefore, we should consider, besides the class distribution, other characteristics of data, such as noise, that may hamper classification.

— *Improving Identification of Difficult Small Classes by Balancing Class Distribution*, 2001.

This approach was proposed by Jorma Laurikkala in her 2001 paper titled *Improving Identification of Difficult Small Classes by Balancing Class Distribution*. The approach involves first selecting all examples from the minority class. Then all of the ambiguous examples in the majority class are identified using the ENN rule and removed. Finally, a one-step version of CNN is used where those remaining examples in the majority class that are misclassified against the store are removed, but only if the number of examples in the majority class is larger than half the size of the minority class.

-
1. Split data T into the class of interest C and the rest of data O .
 2. Identify noisy data A_1 in O with edited nearest neighbor rule.
 3. For each class C_i in O
 - if ($x \in C_i$ in 3-nearest neighbors of misclassified $y \in C$)
 - and ($|C_i| < 0.5 \cdot |C|$) then $A_2 = \{x\} \cup A_2$
 4. Reduced data $S = T - (A_1 \cup A_2)$
-

Figure 13.10: Summary of the Neighborhood Cleaning Rule Algorithm. Taken from *Improving Identification of Difficult Small Classes by Balancing Class Distribution*

This technique can be implemented using the `NeighbourhoodCleaningRule` imbalanced-learn class. The number of neighbors used in the ENN and CNN steps can be specified via the `n_neighbors` argument that defaults to three. The `threshold_cleaning` controls whether or not the CNN is applied to a given class, which might be useful if there are multiple minority classes with similar sizes. This is kept at 0.5.

The complete example of applying NCR on the binary classification problem is listed below. Given the focus on data cleaning over removing redundant examples, we would expect only a modest reduction in the number of examples in the majority class.

```
# undersample and plot imbalanced dataset with the neighborhood cleaning rule
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.under_sampling import NeighbourhoodCleaningRule
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# summarize class distribution
counter = Counter(y)
print(counter)
# define the undersampling method
undersample = NeighbourhoodCleaningRule(n_neighbors=3, threshold_cleaning=0.5)
# transform the dataset
X, y = undersample.fit_resample(X, y)
# summarize the new class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Listing 13.21: Example of applying the NCR strategy to the imbalanced classification dataset.

Running the example first reports the class distribution in the raw dataset, then the transformed dataset. We can see that only 114 examples from the majority class were removed.

```
Counter({0: 9900, 1: 100})
Counter({0: 9786, 1: 100})
```

Listing 13.22: Example output from applying the NCR strategy to the imbalanced classification dataset.

Given the limited and focused amount of undersampling performed, the change to the mass of majority examples is not obvious from the scatter plot that is created.

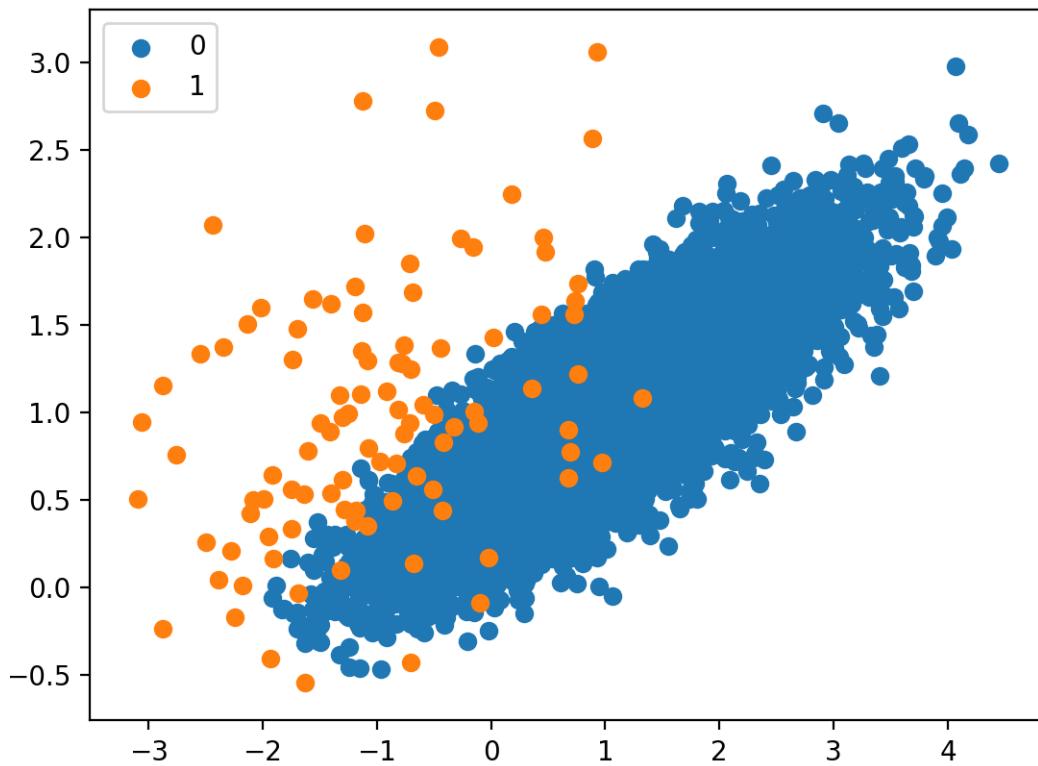


Figure 13.11: Scatter Plot of Imbalanced Dataset Undersampled With the Neighborhood Cleaning Rule.

13.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

13.6.1 Papers

- *KNN Approach To Unbalanced Data Distributions: A Case Study Involving Information Extraction*, 2003.
<https://www.site.uottawa.ca/~nat/Workshop2003/jzhang.pdf>
- *The Condensed Nearest Neighbor Rule (Corresp.)*, 1968.
<https://ieeexplore.ieee.org/document/1054155>
- *Two Modifications of CNN*, 1976.
<https://ieeexplore.ieee.org/document/4309452>
- *Addressing The Curse Of Imbalanced Training Sets: One-sided Selection*, 1997.
<https://sci2s.ugr.es/keel/pdf/algorithm/congreso/kubat97addressing.pdf>

- *Asymptotic Properties of Nearest Neighbor Rules Using Edited Data*, 1972.
<https://ieeexplore.ieee.org/document/4309137>
- *An Experiment with the Edited Nearest-Neighbor Rule*, 1976.
<https://ieeexplore.ieee.org/document/4309523>
- *Improving Identification of Difficult Small Classes by Balancing Class Distribution*, 2001.
https://link.springer.com/chapter/10.1007%2F3-540-48229-6_9

13.6.2 Books

- *Learning from Imbalanced Data Sets*, 2018.
<https://amzn.to/307Xlva>
- *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.
<https://amzn.to/32K9K6d>

13.6.3 API

- Under-sampling, Imbalanced-Learn User Guide.
https://imbalanced-learn.org/stable/under_sampling.html
- imblearn.under_sampling.NearMiss API.
https://imbalanced-learn.org/stable/generated/imblearn.under_sampling.NearMiss.html
- imblearn.under_sampling.CondensedNearestNeighbour API.
https://imbalanced-learn.org/stable/generated/imblearn.under_sampling.CondensedNearestNeighbour.html
- imblearn.under_sampling.TomekLinks API.
https://imbalanced-learn.org/stable/generated/imblearn.under_sampling.TomekLinks.html
- imblearn.under_sampling.OneSidedSelection API.
https://imbalanced-learn.org/stable/generated/imblearn.under_sampling.OneSidedSelection.html
- imblearn.under_sampling.EditedNearestNeighbours API..
https://imbalanced-learn.org/stable/generated/imblearn.under_sampling.EditedNearestNeighbours.html
- imblearn.under_sampling.NeighbourhoodCleaningRule API.
https://imbalanced-learn.org/stable/generated/imblearn.under_sampling.NeighbourhoodCleaningRule.html

13.6.4 Articles

- Oversampling and undersampling in data analysis, Wikipedia.
https://en.wikipedia.org/wiki/Oversampling_and_undersampling_in_data_analysis

13.7 Summary

In this tutorial, you discovered undersampling methods for imbalanced classification. Specifically, you learned:

- How to use the Near-Miss and Condensed Nearest Neighbor Rule methods that select examples to keep in the majority class.
- How to use Tomek Links and the Edited Nearest Neighbors Rule methods that select examples to delete from the majority class.
- How to use One-Sided Selection and the Neighborhood Cleaning Rule that combine methods for choosing examples to keep and delete from the majority class.

13.7.1 Next

In the next tutorial, you will discover how to combine oversampling and undersampling techniques to change the distribution of the training dataset.

Chapter 14

Oversampling and Undersampling

Data sampling methods are designed to add or remove examples from the training dataset in order to change the class distribution. Once the class distributions are more balanced, the suite of standard machine learning classification algorithms can be fit successfully on the transformed datasets.

Oversampling methods duplicate or create new synthetic examples in the minority class, whereas undersampling methods delete examples in the majority class. Both types of sampling can be effective when used in isolation, although can be more effective when both types of methods are used together. In this tutorial, you will discover how to combine oversampling and undersampling techniques for imbalanced classification. After completing this tutorial, you will know:

- How to define a sequence of oversampling and undersampling methods to be applied to a training dataset or when evaluating a classifier model.
- How to manually combine oversampling and undersampling methods for imbalanced classification.
- How to use pre-defined and well-performing combinations of sampling methods for imbalanced classification.

Let's get started.

Note: This chapter makes use of the imbalanced-learn library. See Appendix [B](#) for installation instructions, if needed.

14.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Binary Test Problem and Decision Tree Model
2. Manually Combine Data Sampling Methods
3. Standard Combined Data Sampling Methods

14.2 Binary Test Problem and Decision Tree Model

Before we dive into combinations of oversampling and undersampling methods, let's define a synthetic dataset and model. We can define a synthetic binary classification dataset using the `make_classification()` function from the scikit-learn library. For example, we can create 10,000 examples with two input variables and a 1:100 class distribution as follows:

```
...
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
```

Listing 14.1: Example of defining an imbalanced binary classification problem.

We can then create a scatter plot of the dataset via the `scatter()` Matplotlib function to understand the spatial relationship of the examples in each class and their imbalance.

```
...
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Listing 14.2: Example of creating a scatter plot with dots colored by class label.

Tying this together, the complete example of creating an imbalanced classification dataset and plotting the examples is listed below.

```
# Generate and plot a synthetic imbalanced classification dataset
from collections import Counter
from sklearn.datasets import make_classification
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# summarize class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Listing 14.3: Example of defining and summarizing an imbalanced classification dataset.

Running the example first summarizes the class distribution, showing an approximate 1:100 class distribution with about 10,000 examples with class 0 and 100 with class 1.

```
Counter({0: 9900, 1: 100})
```

Listing 14.4: Example output from defining and summarizing an imbalanced classification dataset.

Next, a scatter plot is created showing all of the examples in the dataset. We can see a large mass of examples for class 0 (blue) and a small number of examples for class 1 (orange). We can also see that the classes overlap with some examples from class 1 clearly within the part of the feature space that belongs to class 0.

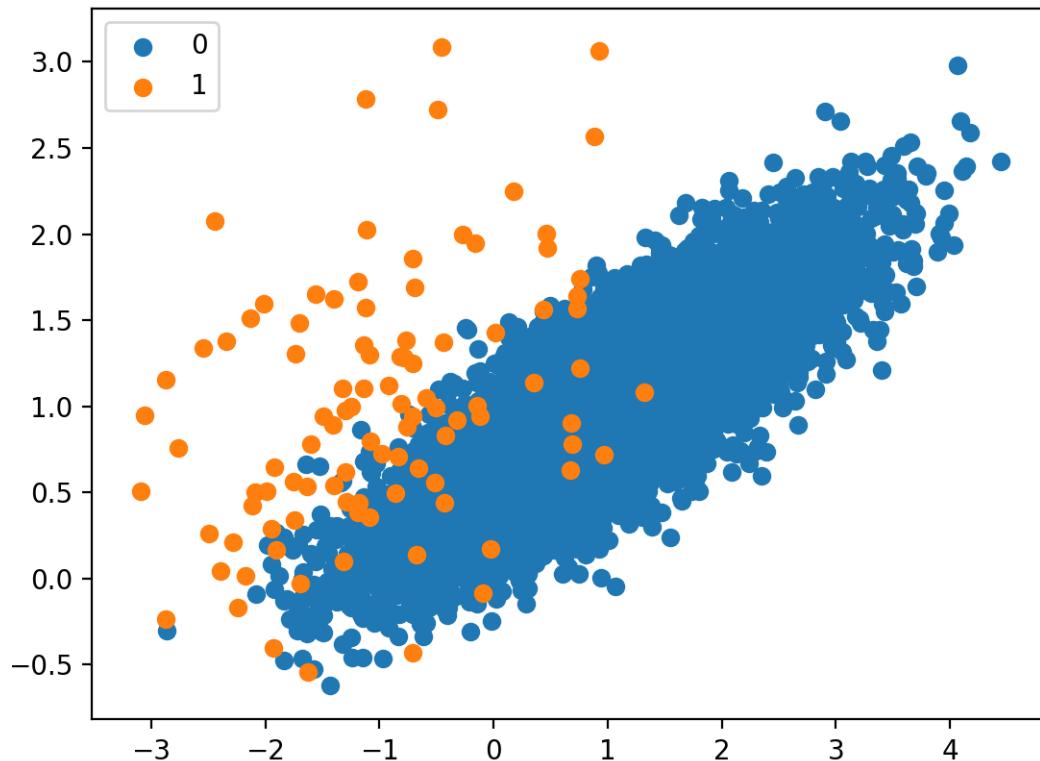


Figure 14.1: Scatter Plot of Imbalanced Classification Dataset.

We can fit a `DecisionTreeClassifier` model on this dataset. It is a good model to test because it is sensitive to the class distribution in the training dataset.

```
...
# define model
model = DecisionTreeClassifier()
```

Listing 14.5: Example of defining a classification model.

We can evaluate the model using repeated stratified k -fold cross-validation with three repeats and 10 folds. The ROC area under curve (AUC) measure can be used to estimate the performance of the model. It can be optimistic for severely imbalanced datasets, although it does correctly show relative improvements in model performance.

```
...
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
```

```

scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))

```

Listing 14.6: Example of defining a model evaluation model.

Tying this together, the example below evaluates a decision tree model on the imbalanced classification dataset.

```

# evaluates a decision tree model on the imbalanced dataset
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
# generate 2 class dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# define model
model = DecisionTreeClassifier()
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))

```

Listing 14.7: Example of evaluating a classification model on the imbalanced dataset.

Running the example reports the average ROC AUC for the decision tree on the dataset over three repeats of 10-fold cross-validation (e.g. average over 30 different model evaluations).

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this example, you can see that the model achieved a ROC AUC of about 0.76. This provides a baseline on this dataset, which we can use to compare different combinations of over and under sampling methods on the training dataset.

```
Mean ROC AUC: 0.762
```

Listing 14.8: Example output from evaluating a classification model on the imbalanced dataset.

Now that we have a test problem, model, and test harness, let's look at manual combinations of oversampling and undersampling methods.

14.3 Manually Combine Data Sampling Methods

The imbalanced-learn Python library provides a range of sampling techniques, as well as a `Pipeline` class that can be used to create a combined sequence of sampling methods to apply to a dataset. We can use the `Pipeline` to construct a sequence of oversampling and undersampling techniques to apply to a dataset. For example:

```
...
# define sampling
over = ...
under = ...
# define pipeline
pipeline = Pipeline(steps=[('o', over), ('u', under)])
```

Listing 14.9: Example of defining a `Pipeline` of oversampling and undersampling methods.

This pipeline first applies an oversampling technique to a dataset, then applies undersampling to the output of the oversampling transform before returning the final outcome. It allows transforms to be stacked or applied in sequence on a dataset. The pipeline can then be used to transform a dataset; for example:

```
...
# fit and apply the pipeline
X_resampled, y_resampled = pipeline.fit_resample(X, y)
```

Listing 14.10: Example of applying a `Pipeline` of data sampling methods.

Alternately, a model can be added as the last step in the pipeline. This allows the pipeline to be treated as a model. When it is fit on a training dataset, the transforms are first applied to the training dataset, then the transformed dataset is provided to the model so that it can develop a fit.

```
...
# define model
model = ...
# define sampling
over = ...
under = ...
# define pipeline
pipeline = Pipeline(steps=[('o', over), ('u', under), ('m', model)])
```

Listing 14.11: Example of defining a `Pipeline` of data sampling and a model.

Recall that the sampling is only applied to the training dataset, not the test dataset. When used in k -fold cross-validation, the entire sequence of transforms and fit is applied on each training dataset comprised of cross-validation folds. This is important as both the transforms and fit are performed without knowledge of the holdout set, which avoids data leakage. For example:

```
...
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(pipeline, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
```

Listing 14.12: Example of evaluating a `Pipeline` of data sampling and a model.

Now that we know how to manually combine sampling methods, let's look at two examples.

14.3.1 Random Oversampling and Undersampling

A good starting point for combining sampling techniques is to start with random or naive methods. Although they are simple, and often ineffective when applied in isolation, they can be

effective when combined. Random oversampling involves randomly duplicating examples in the minority class, whereas random undersampling involves randomly deleting examples from the majority class.

As these two transforms are performed on separate classes, the order in which they are applied to the training dataset does not matter. The example below defines a pipeline that first oversamples the minority class to 10 percent of the majority class, under samples the majority class more than the minority class, and then fits a decision tree model.

```
...
# define model
model = DecisionTreeClassifier()
# define sampling
over = RandomOverSampler(sampling_strategy=0.1)
under = RandomUnderSampler(sampling_strategy=0.5)
# define pipeline
pipeline = Pipeline(steps=[('o', over), ('u', under), ('m', model)])
```

Listing 14.13: Example of random oversampling and random undersampling.

The complete example of evaluating this combination on the binary classification problem is listed below.

```
# combination of random oversampling and undersampling for imbalanced classification
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.tree import DecisionTreeClassifier
from imblearn.pipeline import Pipeline
from imblearn.over_sampling import RandomOverSampler
from imblearn.under_sampling import RandomUnderSampler
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# define model
model = DecisionTreeClassifier()
# define sampling
over = RandomOverSampler(sampling_strategy=0.1)
under = RandomUnderSampler(sampling_strategy=0.5)
# define pipeline
pipeline = Pipeline(steps=[('o', over), ('u', under), ('m', model)])
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(pipeline, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))
```

Listing 14.14: Example of evaluating random oversampling and random undersampling with a classification model.

Running the example evaluates the system of transforms and the model and summarizes the performance as the mean ROC AUC.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see a modest lift in ROC AUC performance from 0.76 with no transforms to about 0.81 with random over- and undersampling.

```
Mean ROC AUC: 0.814
```

Listing 14.15: Example output from evaluating random oversampling and random undersampling with a classification model.

14.3.2 SMOTE and Random Undersampling

We are not limited to using random sampling methods. Perhaps the most popular oversampling method is the Synthetic Minority Oversampling Technique, or SMOTE for short. SMOTE works by selecting examples that are close in the feature space, drawing a line between the examples in the feature space and drawing a new sample as a point along that line. The authors of the technique recommend using SMOTE on the minority class, followed by an undersampling technique on the majority class.

The combination of SMOTE and under-sampling performs better than plain under-sampling.

— SMOTE: *Synthetic Minority Over-sampling Technique*, 2011.

We can combine SMOTE with `RandomUnderSampler`. Again, the order in which these procedures are applied does not matter as they are performed on different subsets of the training dataset. The pipeline below implements this combination, first applying SMOTE to bring the minority class distribution to 10 percent of the majority class, then using `RandomUnderSampler` to bring the minority class down to a fraction larger than the minority class before fitting a `DecisionTreeClassifier`.

```
...
# define model
model = DecisionTreeClassifier()
# define pipeline
over = SMOTE(sampling_strategy=0.1)
under = RandomUnderSampler(sampling_strategy=0.5)
steps = [('o', over), ('u', under), ('m', model)]
```

Listing 14.16: Example of SMOTE oversampling and random undersampling.

The example below evaluates this combination on our imbalanced binary classification problem.

```
# combination of SMOTE and random undersampling for imbalanced classification
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.tree import DecisionTreeClassifier
from imblearn.pipeline import Pipeline
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
# generate dataset
```

```

X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# define model
model = DecisionTreeClassifier()
# define pipeline
over = SMOTE(sampling_strategy=0.1)
under = RandomUnderSampler(sampling_strategy=0.5)
steps = [('o', over), ('u', under), ('m', model)]
pipeline = Pipeline(steps=steps)
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(pipeline, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))

```

Listing 14.17: Example of evaluating SMOTE and random undersampling with a classification model.

Running the example evaluates the system of transforms and the model and summarizes the performance as the mean ROC AUC.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see another lift in ROC AUC performance from about 0.81 to about 0.83.

```
Mean ROC AUC: 0.833
```

Listing 14.18: Example output from evaluating SMOTE and random undersampling with a classification model.

14.4 Standard Combined Data Sampling Methods

There are combinations of oversampling and undersampling methods that have proven effective and together may be considered sampling techniques. Two examples are the combination of SMOTE with Tomek Links undersampling and SMOTE with Edited Nearest Neighbors undersampling. The imbalanced-learn Python library provides implementations for both of these combinations directly. Let's take a closer look at each in turn.

14.4.1 SMOTE and Tomek Links Undersampling

SMOTE is an oversampling method that synthesizes new plausible examples in the minority class. Tomek Links refers to a method for identifying pairs of nearest neighbors in a dataset that have different classes. Removing one or both of the examples in these pairs (such as the examples in the majority class) has the effect of making the decision boundary in the training dataset less noisy or ambiguous.

Gustavo Batista, et al. tested combining these methods in their 2003 paper titled *Balancing Training Data for Automated Annotation of Keywords: A Case Study*. Specifically, first the

SMOTE method is applied to oversample the minority class to a balanced distribution, then examples in Tomek Links from the majority classes are identified and removed.

In this work, only majority class examples that participate in a Tomek link were removed, since minority class examples were considered too rare to be discarded. [...] In our work, as minority class examples were artificially created and the data sets are currently balanced, then both majority and minority class examples that form a Tomek link, are removed.

— *Balancing Training Data for Automated Annotation of Keywords: A Case Study*, 2003.

The combination was shown to provide a reduction in false negatives at the cost of an increase in false positives for a binary classification task. We can implement this combination using the `SMOTETomek` class.

```
...
# define sampling
resample = SMOTETomek()
```

Listing 14.19: Example of SMOTE oversampling and Tomek Links undersampling.

The SMOTE configuration can be set via the `smote` argument and takes a configured SMOTE instance. The Tomek Links configuration can be set via the `tomek` argument and takes a configured `TomekLinks` object. Both arguments are set to instances of each class with default configurations. The default is to balance the dataset with SMOTE then remove Tomek links from all classes. This is the approach used in another paper that explores this combination titled *A Study of the Behavior of Several Methods for Balancing Machine Learning Training Data*.

... we propose applying Tomek links to the over-sampled training set as a data cleaning method. Thus, instead of removing only the majority class examples that form Tomek links, examples from both classes are removed.

— *A Study of the Behavior of Several Methods for Balancing Machine Learning Training Data*, 2004.

Alternately, we can configure the combination to only remove links from the majority class as described in the 2003 paper by specifying the `tomek` argument with an instance of `TomekLinks` with the `sampling_strategy` argument set to only undersample the ‘`majority`’ class; for example:

```
...
# define sampling
resample = SMOTETomek(tomek=TomekLinks(sampling_strategy='majority'))
```

Listing 14.20: Example of SMOTE oversampling and Tomek Links undersampling with a customized configuration.

We can evaluate this combined sampling strategy with a decision tree classifier on our binary classification problem. The complete example is listed below.

```

# combined SMOTE and Tomek Links sampling for imbalanced classification
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from imblearn.pipeline import Pipeline
from sklearn.tree import DecisionTreeClassifier
from imblearn.combine import SMOTETomek
from imblearn.under_sampling import TomekLinks
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# define model
model = DecisionTreeClassifier()
# define sampling
resample = SMOTETomek(tomek=TomekLinks(sampling_strategy='majority'))
# define pipeline
pipeline = Pipeline(steps=[('r', resample), ('m', model)])
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(pipeline, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))

```

Listing 14.21: Example of evaluating the SMOTETomek strategy with a classification model.

Running the example evaluates the system of transforms and the model and summarizes the performance as the mean ROC AUC.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, it seems that this combined sampling strategy does not offer a benefit for this model on this dataset.

Mean ROC AUC: 0.815

Listing 14.22: Example output from evaluating the SMOTETomek strategy with a classification model.

14.4.2 SMOTE and Edited Nearest Neighbors Undersampling

SMOTE may be the most popular oversampling technique and can be combined with many different undersampling techniques. Another very popular undersampling method is the Edited Nearest Neighbors, or ENN, rule. This rule involves using $k = 3$ nearest neighbors to locate those examples in a dataset that are misclassified and that are then removed. It can be applied to all classes or just those examples in the majority class. Gustavo Batista, et al. explore many combinations of oversampling and undersampling methods compared to the methods used in isolation in their 2004 paper titled *A Study of the Behavior of Several Methods for Balancing Machine Learning Training Data*. This includes the combinations:

- Condensed Nearest Neighbors + Tomek Links
- SMOTE + Tomek Links
- SMOTE + Edited NearestNeighbors

Regarding this final combination, the authors comment that ENN is more aggressive at downampling the majority class than Tomek Links, providing more in-depth cleaning. They apply the method, removing examples from both the majority and minority classes.

... ENN is used to remove examples from both classes. Thus, any example that is misclassified by its three nearest neighbors is removed from the training set.

— *A Study of the Behavior of Several Methods for Balancing Machine Learning Training Data*, 2004.

This can be implemented via the `SMOTEENN` class in the `imbalanced-learn` library.

```
...
# define sampling
resample = SMOTEENN()
```

Listing 14.23: Example of SMOTE oversampling and ENN undersampling.

The SMOTE configuration can be set as a `SMOTE` object via the `smote` argument, and the ENN configuration can be set via the `EditedNearestNeighbours` object via the `enn` argument. SMOTE defaults to balancing the distribution, followed by ENN that by default removes misclassified examples from all classes. We could change the ENN to only remove examples from the majority class by setting the `enn` argument to an `EditedNearestNeighbours` instance with `sampling_strategy` argument set to ‘`majority`’.

```
...
# define sampling
resample = SMOTEENN(enn=EditedNearestNeighbours(sampling_strategy='majority'))
```

Listing 14.24: Example of SMOTE oversampling and ENN undersampling with a customized configuration.

We can evaluate the default strategy (editing examples in all classes) and evaluate it with a decision tree classifier on our imbalanced dataset. The complete example is listed below.

```
# combined SMOTE and Edited Nearest Neighbors sampling for imbalanced classification
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from imblearn.pipeline import Pipeline
from sklearn.tree import DecisionTreeClassifier
from imblearn.combine import SMOTEENN
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# define model
model = DecisionTreeClassifier()
# define sampling
```

```

resample = SMOTEENN()
# define pipeline
pipeline = Pipeline(steps=[('r', resample), ('m', model)])
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(pipeline, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))

```

Listing 14.25: Example of evaluating the SMOTEENN strategy with a classification model.

Running the example evaluates the system of transforms and the model and summarizes the performance as the mean ROC AUC.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we see a further lift in performance over SMOTE with the random undersampling method from about 0.81 to about 0.85.

```
Mean ROC AUC: 0.856
```

Listing 14.26: Example output from evaluating the SMOTEENN strategy with a classification model.

This result highlights that editing the oversampled minority class may also be an important consideration that could easily be overlooked. This was the same finding in the 2004 paper where the authors discover that SMOTE with Tomek Links and SMOTE with ENN perform well across a range of datasets.

Our results show that the over-sampling methods in general, and Smote + Tomek and Smote + ENN (two of the methods proposed in this work) in particular for data sets with few positive (minority) examples, provided very good results in practice.

— *A Study of the Behavior of Several Methods for Balancing Machine Learning Training Data*, 2004.

14.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

14.5.1 Papers

- *SMOTE: Synthetic Minority Over-sampling Technique*, 2011.
<https://arxiv.org/abs/1106.1813>
- *Balancing Training Data for Automated Annotation of Keywords: a Case Study*, 2003.
<http://www.inf.ufrgs.br/maslab/pergamus/pubs/balancing-training-data-for.pdf>
- *A Study of the Behavior of Several Methods for Balancing Machine Learning Training Data*, 2004.
<https://dl.acm.org/citation.cfm?id=1007735>

14.5.2 Books

- *Learning from Imbalanced Data Sets*, 2018.
<https://amzn.to/307Xlva>
- *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.
<https://amzn.to/32K9K6d>

14.5.3 API

- imbalanced-learn, GitHub.
<https://github.com/scikit-learn-contrib/imbalanced-learn>
- Combination of over- and under-sampling, Imbalanced Learn User Guide.
<https://imbalanced-learn.org/stable/combine.html>
- imblearn.over_sampling.RandomOverSampler API.
https://imbalanced-learn.org/stable/generated/imblearn.over_sampling.RandomOverSampler.html
- imblearn.pipeline.Pipeline API.
<https://imbalanced-learn.org/stable/generated/imblearn.pipeline.Pipeline.html>
- imblearn.under_sampling.RandomUnderSampler API.
https://imbalanced-learn.org/stable/generated/imblearn.under_sampling.RandomUnderSampler.html
- imblearn.over_sampling.SMOTE API.
https://imbalanced-learn.org/stable/generated/imblearn.over_sampling.SMOTE.html
- imblearn.combine.SMOTEToTomek API.
<https://imbalanced-learn.org/stable/generated/imblearn.combine.SMOTEToTomek.html>
- imblearn.combine.SMOTETENN API.
<https://imbalanced-learn.org/stable/generated/imblearn.combine.SMOTETENN.html>

14.5.4 Articles

- Oversampling and undersampling in data analysis, Wikipedia.
https://en.wikipedia.org/wiki/Oversampling_and_undersampling_in_data_analysis

14.6 Summary

In this tutorial, you discovered how to combine oversampling and undersampling techniques for imbalanced classification. Specifically, you learned:

- How to define a sequence of oversampling and undersampling methods to be applied to a training dataset or when evaluating a classifier model.

- How to manually combine oversampling and undersampling methods for imbalanced classification.
- How to use pre-defined and well-performing combinations of sampling methods for imbalanced classification.

14.6.1 Next

This was the final tutorial in this Part. In the next Part, you will discover cost-sensitive algorithms that you can use for imbalanced classification.

Part V

Cost-Sensitive

Chapter 15

Cost-Sensitive Learning

Most machine learning algorithms assume that all misclassification errors made by a model are equal. This is often not the case for imbalanced classification problems where missing a positive or minority class case is worse than incorrectly classifying an example from the negative or majority class. There are many real-world examples, such as detecting spam email, diagnosing a medical condition, or identifying fraud. In all of these cases, a false negative (missing a case) is worse or more costly than a false positive.

Cost-sensitive learning is a subfield of machine learning that takes the costs of prediction errors (and potentially other costs) into account when training a machine learning model. It is a field of study that is closely related to the field of imbalanced learning that is concerned with classification on datasets with a skewed class distribution. As such, many conceptualizations and techniques developed and used for cost-sensitive learning can be adopted for imbalanced classification problems. In this tutorial, you will discover a gentle introduction to cost-sensitive learning for imbalanced classification. After completing this tutorial, you will know:

- Imbalanced classification problems often value false-positive classification errors differently from false negatives.
- Cost-sensitive learning is a subfield of machine learning that involves explicitly defining and using costs when training machine learning algorithms.
- Cost-sensitive techniques may be divided into three groups, including data sampling, algorithm modifications, and ensemble methods.

Let's get started.

15.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Not All Classification Errors Are Equal
2. Cost-Sensitive Learning
3. Cost-Sensitive Imbalanced Classification
4. Cost-Sensitive Methods

15.2 Not All Classification Errors Are Equal

Classification is a predictive modeling problem that involves predicting the class label for an observation. There may be many class labels, so-called multiclass classification problems, although the simplest and perhaps most common type of classification problem has two classes and is referred to as binary classification. Most machine learning algorithms designed for classification assume that there is an equal number of examples for each observed class. This is not always the case in practice, and datasets that have a skewed class distribution are referred to as imbalanced classification problems.

In cost-sensitive learning instead of each instance being either correctly or incorrectly classified, each class (or instance) is given a misclassification cost. Thus, instead of trying to optimize the accuracy, the problem is then to minimize the total misclassification cost.

— Page 50, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

In addition to assuming that the class distribution is balanced, most machine learning algorithms also assume that the prediction errors made by a classifier are the same, so-called miss-classifications. This is typically not the case for binary classification problems, especially those that have an imbalanced class distribution.

Most classifiers assume that the misclassification costs (false negative and false positive cost) are the same. In most real-world applications, this assumption is not true.

— *Cost-sensitive Learning Methods For Imbalanced Data*, 2010.

For imbalanced classification problems, the examples from the majority class are referred to as the negative class and assigned the class label 0. Those examples from the minority class are referred to as the positive class and are assigned the class label 1. The reason for this negative vs. positive naming convention is because the examples from the majority class typically represent a normal or no-event case, whereas examples from the minority class represent the exceptional or event case.

- **Majority Class:** Negative or no-event assigned the class label 0.
- **Minority Class:** Positive or event assigned the class label 1.

Real-world imbalanced binary classification problems typically have a different interpretation for each of the classification errors that can be made. For example, classifying a negative case as a positive case is typically far less of a problem than classifying a positive case as a negative case. This makes sense if we consider the goal of a classifier on imbalanced binary classification problems is to detect the positive cases correctly and positive cases represent an exceptional or event that we are most interested in. We can make this clear with some examples.

Bank Loan Problem

Consider a problem where a bank wants to determine whether to give a loan to a customer or not. Denying a loan to a good customer is not as bad as giving a loan to a bad customer that may never repay it.

Cancer Diagnosis Problem

Consider a problem where a doctor wants to determine whether a patient has cancer or not. It is better to diagnose a healthy patient with cancer and follow-up with more medical tests than it is to discharge a patient that has cancer.

... in medical diagnosis of a certain cancer, if the cancer is regarded as the positive class, and non-cancer (healthy) as negative, then missing a cancer (the patient is actually positive but is classified as negative; thus it is also called “false negative”) is much more serious (thus expensive) than the false-positive error.

— *Cost-Sensitive Learning, Encyclopedia of Machine Learning, 2010.*

Fraud Detection Problem

Consider the problem of an insurance company wants to determine whether a claim is fraudulent. Identifying good claims as fraudulent and following up with the customer is better than honoring fraudulent insurance claims.

We can see with these examples that misclassification errors are not desirable in general, but one type of misclassification is much worse than the other. Specifically predicting positive cases as a negative case is more harmful, more expensive, or worse in whatever way we want to measure the context of the target domain.

... it is often more expensive to misclassify an actual positive example into negative, than an actual negative example into positive.

— *Cost-Sensitive Learning, Encyclopedia of Machine Learning, 2010.*

Machine learning algorithms that treat each type of misclassification error as the same are unable to meet the needs of these types of problems. As such, both the underrepresentation of the minority class in the training data and the increased importance on correctly identifying examples from the minority class make imbalanced classification one of the most challenging problems in applied machine learning.

Class imbalance is one of the challenging problems for machine learning algorithms.

— *Cost-sensitive Learning Methods For Imbalanced Data, 2010.*

15.3 Cost-Sensitive Learning

There is a subfield of machine learning that is focused on learning and using models on data that have uneven penalties or costs when making predictions and more. This field is generally referred to as Cost-Sensitive Machine Learning, or more simply Cost-Sensitive Learning.

... the machine learning algorithm needs to be sensitive to the cost that it is dealing with, and in the better case take the cost into account in the model fitting process. This leads to cost-sensitive machine learning, a relatively new research topic in machine learning.

— Page xiii, *Cost-Sensitive Machine Learning*, 2011.

Traditionally, machine learning algorithms are trained on a dataset and seek to minimize error. Fitting a model on data solves an optimization problem where we explicitly seek to minimize error. A range of functions can be used to calculate the error of a model on training data, and the more general term is referred to as loss. We seek to minimize the loss of a model on the training data, which is the same as talking about error minimization.

- **Error Minimization:** The conventional goal when training a machine learning algorithm is to minimize the error of the model on a training dataset.

In cost-sensitive learning, a penalty associated with an incorrect prediction and is referred to as a *cost*. We could alternately refer to the inverse of the penalty as the *benefit*, although this framing is rarely used.

- **Cost:** The penalty associated with an incorrect prediction.

The goal of cost-sensitive learning is to minimize the cost of a model on the training dataset, where it is assumed that different types of prediction errors have a different and known associated cost.

- **Cost Minimization:** The goal of cost-sensitive learning is to minimize the cost of a model on a training dataset.

Cost-Sensitive Learning is a type of learning that takes the misclassification costs (and possibly other types of cost) into consideration. The goal of this type of learning is to minimize the total cost.

— *Cost-sensitive Learning Methods For Imbalanced Data*, 2010.

There is a tight-coupling between imbalanced classification and cost-sensitive learning. Specifically, an imbalanced learning problem can be addressed using cost-sensitive learning. Nevertheless, cost-sensitive learning is a separate subfield of study and cost may be defined more broadly than prediction error or classification error. This means that although some methods from cost-sensitive learning can be helpful on imbalanced classification, not all cost-sensitive learning techniques are imbalanced-learning techniques, and conversely, not all methods used to address imbalanced learning are appropriate for cost-sensitive learning.

To make this concrete, we can consider a wide range of other ways we might wish to consider or measure cost when training a model on a dataset. For example, Peter Turney lists nine types of costs that might be considered in machine learning in his 2000 paper titled *Types of Cost in Inductive Concept Learning*. In summary, they are:

- Cost of misclassification errors (or prediction errors more generally).
- Cost of tests or evaluation.
- Cost of labeling.
- Cost of intervention or changing the system from which observations are drawn.

- Cost of unwanted achievements or outcomes from intervening.
- Cost of computation or computational complexity.
- Cost of cases or data collection.
- Cost of human-computer interaction or framing the problem and using software to fit and use a model.
- Cost of instability or variance known as concept drift.

Although critical to many real-world problems, the idea of costs and cost-sensitive learning is a new topic that was largely ignored up until recently.

In real-world applications of concept learning, there are many different types of cost involved. The majority of the machine learning literature ignores all types of cost ...

— *Types of Cost in Inductive Concept Learning*, 2000.

The above list highlights that the cost we are interested in for imbalanced-classification is just one type of the range of costs that the broader field of cost-sensitive learning might consider. In the next section, we will take a closer look at how we can harness ideas of misclassification cost-sensitive learning to help with imbalanced classification.

15.4 Cost-Sensitive Imbalanced Classification

Cost-sensitive learning for imbalanced classification is focused on first assigning different costs to the types of misclassification errors that can be made, then using specialized methods to take those costs into account. The varying misclassification costs are best understood using the idea of a cost matrix. Let's start by reviewing the confusion matrix.

A confusion matrix is a summary of the predictions made by a model on classification tasks. It is a table that summarizes the number of predictions made for each class, separated by the actual class to which each example belongs.

It is best understood using a binary classification problem with negative and positive classes, typically assigned 0 and 1 class labels respectively. The columns of the table represent the actual class to which examples belong, and the rows represent the predicted class (although the meaning of rows and columns can and often are interchanged with no loss of meaning). A cell in the table is the count of the number of examples that meet the conditions of the row and column, and each cell has a specific common name.

An example of a confusion matrix for a binary classification task is listed below showing the common names for the values in each of the four cells of the table.

	Actual Negative	Actual Positive
Predicted Negative	True Negative	False Negative
Predicted Positive	False Positive	True Positive

Listing 15.1: Binary Confusion Matrix.

We can see that we are most interested in the errors, the so-called False Positives and False Negatives, and it is the False Negatives that probably interest us the most on many imbalanced classification tasks. Now, we can consider the same table with the same rows and columns and assign a cost to each of the cells. This is called a cost matrix.

- **Cost Matrix:** A matrix that assigns a cost to each cell in the confusion matrix.

The example below is a cost matrix where we use the notation $C()$ to indicate the cost, the first term in parenthesis represents the predicted class and the second term represents the actual class. The names of each cell from the confusion matrix are also listed as acronyms, e.g. False Positive is FP.

	Actual Negative	Actual Positive
Predicted Negative	$C(0,0)$, TN	$C(0,1)$, FN
Predicted Positive	$C(1,0)$, FP	$C(1,1)$, TP

Listing 15.2: Binary Cost Matrix.

We can see that the cost of a False Positive is $C(1,0)$ and the cost of a False Negative is $C(0,1)$. This formulation and notation of the cost matrix comes from Charles Elkan's seminal 2001 paper on the topic titled *The Foundations of Cost-Sensitive Learning*. An intuition from this matrix is that the cost of misclassification is always higher than correct classification, otherwise, cost can be minimized by predicting one class.

Conceptually, the cost of labeling an example incorrectly should always be greater than the cost of labeling it correctly.

— *The Foundations Of Cost-sensitive Learning*, 2001.

For example, we might assign no cost to correct predictions in each class, a cost of 5 for False Positives and a cost of 88 for False Negatives.

	Actual Negative	Actual Positive
Predicted Negative	0	88
Predicted Positive	5	0

Listing 15.3: Example of a Specific Binary Cost Matrix.

We can define the total cost of a classifier using this framework as the cost-weighted sum of the False Negatives and False Positives.

$$\text{TotalCost} = C(0,1) \times \text{FalseNegatives} + C(1,0) \times \text{FalsePositives} \quad (15.1)$$

This is the value that we seek to minimize in cost-sensitive learning, at least conceptually.

The purpose of CSL is to build a model with minimum misclassification costs (total cost).

— *Cost-sensitive Learning Methods For Imbalanced Data*, 2010.

The values of the cost matrix must be carefully defined. Like the choice of error function for traditional machine learning models, the choice of costs or cost function will determine the quality and utility of the model that is fit on the training data.

The effectiveness of cost-sensitive learning relies strongly on the supplied cost matrix. Parameters provided there will be of crucial importance to both training and predictions steps.

— Page 66, *Learning from Imbalanced Data Sets*, 2018.

In some problem domains, defining the cost matrix might be obvious. In an insurance claim example, the costs for a false positive might be the monetary cost of follow-up with the customer to the company and the cost of a false negative might be the cost of the insurance claim. In other domains, defining the cost matrix might be challenging. For example, in a cancer diagnostic test example, the cost of a false positive might be the monetary cost of performing subsequent tests, whereas what is the equivalent dollar cost for letting a sick patient go home and get sicker? A cost matrix might be able to be defined by a domain expert or economist in such cases, or not.

Further, the cost might be a complex multi-dimensional function, including monetary costs, reputation costs, and more. A good starting point for imbalanced classification tasks is to assign costs based on the inverse class distribution.

In many cases we do not have access to a domain expert and no a priori information on the cost matrix is available during classifier training. This is a common scenario when we want to apply cost-sensitive learning as a method for solving imbalanced problems ...

— Page 67, *Learning from Imbalanced Data Sets*, 2018.

For example, we may have a dataset with a 1 to 100 (1:100) ratio of examples in the minority class to examples in the majority class. This ratio can be inverted and used as the cost of misclassification errors, where the cost of a False Negative is 100 and the cost of a False Positive is 1.

	Actual Negative	Actual Positive
Predicted Negative	0	100
Predicted Positive	1	0

Listing 15.4: Example of a Binary Cost Matrix for Imbalanced Classification.

This is an effective heuristic for setting costs in general, although it assumes that the class distribution observed in the training data is representative of the broader problem and is appropriate for the chosen cost-sensitive method being used. As such, it is a good idea to use this heuristic as a starting point, then test a range of similar related costs or ratios to confirm it is sensible.

15.5 Cost-Sensitive Methods

Cost-sensitive machine learning methods are those that explicitly use the cost matrix. Given our focus on imbalanced classification, we are specifically interested in those cost-sensitive techniques that focus on using varying misclassification costs in some way.

Cost-sensitive learning methods target the problem of imbalanced learning by using different cost matrices that describe the costs for misclassifying any particular data example.

— Page 3-4, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

There are perhaps three main groups of cost-sensitive methods that are most relevant for imbalanced learning; they are:

1. Cost-Sensitive Resampling
2. Cost-Sensitive Algorithms
3. Cost-Sensitive Ensembles

Let's take a closer look at each in turn.

15.5.1 Cost-Sensitive Resampling

In imbalanced classification, data sampling refers to techniques that transform the training dataset to better balance the class distribution. This may involve selectively deleting examples from the majority class, referred to as undersampling. More commonly, it refers to duplicating or synthesizing new examples in the minority class, referred to as oversampling, or combinations of both undersampling and oversampling.

Data sampling is a technique that can be used for cost-sensitive learning directly. Instead of sampling with a focus on balancing the skewed class distribution, the focus is on changing the composition of the training dataset to meet the expectations of the cost matrix. This might involve directly sampling the data distribution or using a method to weight examples in the dataset. Such methods may be referred to as cost-proportionate weighing of the training dataset or cost-proportionate sampling.

We propose and evaluate a family of methods [...] based on cost-proportionate weighting of the training examples, which can be realized either by feeding the weights to the classification algorithm (as often done in boosting), or (in a black box manner) by careful subsampling.

— *Cost-Sensitive Learning by Cost-Proportionate Example Weighting*, 2003.

For imbalanced classification where the cost matrix is defined using the class distribution, there is no difference in the data sampling technique.

15.5.2 Cost-Sensitive Algorithms

Machine learning algorithms are rarely developed specifically for cost-sensitive learning. Instead, the wealth of existing machine learning algorithms can be modified to make use of the cost matrix. This might involve a modification that is unique to each algorithm and which can be quite time consuming to develop and test. Many such algorithm-specific augmentations have been proposed for popular algorithms, like decision trees and support vector machines.

Among all of the classifiers, induction of cost-sensitive decision trees has arguably gained the most attention.

— Page 69, *Learning from Imbalanced Data Sets*, 2018.

The scikit-learn Python machine learning library provides examples of these cost-sensitive extensions via the `class_weight` argument on the following classifiers:

- SVC
- DecisionTreeClassifier

Another more general approach to modifying existing algorithms is to use the costs as a penalty for misclassification when the algorithms are trained. Given that most machine learning algorithms are trained to minimize error, cost for misclassification is added to the error or used to weigh the error during the training process.

This approach can be used for iteratively trained algorithms, such as logistic regression and artificial neural networks. The scikit-learn library provides examples of these cost-sensitive extensions via the `class_weight` argument on the following classifiers:

- LogisticRegression
- RidgeClassifier

The Keras Python Deep Learning library also provides access to this use of cost-sensitive augmentation for neural networks via the `class_weight` argument on the `fit()` function when training models. Again, the line is blurred between cost-sensitive augmentations to algorithms vs. imbalanced classification augmentations to algorithms when the inverse class distribution is used as the cost matrix.

In the domain of cost-sensitive machine learning, these algorithms are referred to with the *Cost-Sensitive* prefix, e.g. *Cost-Sensitive Logistic Regression*, whereas in imbalanced-learning, such algorithms are referred to with a *Class-Weighted* prefix, e.g. *Class-Weighted Logistic Regression* or simply *Weighted Logistic Regression*.

15.5.3 Cost-Sensitive Ensembles

A second distinct group of methods is techniques designed to filter or combine the predictions from traditional machine learning models in order to take misclassification costs into account. These methods are referred to as *wrapper methods* as they wrap a standard machine learning classifier. They are also referred to as *meta-learners* or *ensembles* as they learn how to use or combine predictions from other models.

Cost-sensitive meta-learning converts existing cost-insensitive classifiers into cost-sensitive ones without modifying them. Thus, it can be regarded as a middleware component that pre-processes the training data, or post-processes the output, from the cost-insensitive learning algorithms.

— *Cost-Sensitive Learning, Encyclopedia of Machine Learning*, 2010.

Perhaps the simplest approach is the use of a machine learning model to predict the probability of class membership, then using a line search on the threshold at which examples are assigned to each crisp class label that minimizes the cost of misclassification.

This is often referred to as *thresholding* or threshold optimization and is used more generally for binary classification tasks, although it can easily be modified to minimize cost instead of a specific type of classification error metric. MetaCost is a data preprocessing technique that relabels examples in the training dataset in order to minimize cost.

... we propose a principled method for making an arbitrary classifier cost-sensitive by wrapping a cost-minimizing procedure around it.

— *MetaCost: A General Method for Making Classifiers Cost-Sensitive*, 1999.

In MetaCost, first a bagged ensemble of classifiers is fit on the training dataset in order to identify those examples that need to be relabeled, a transformed version of the dataset with relabeled examples is created, then the ensemble is discarded and the transformed dataset is used to train a classifier model. Another important area is modifications to decision tree ensembles that take the cost matrix into account, such as bagging and boosting algorithms, most notably cost-sensitive versions of AdaBoost such as AdaCost.

AdaCost, a variant of AdaBoost, is a misclassification cost-sensitive boosting method. It uses the cost of misclassifications to update the training distribution on successive boosting rounds.

— *AdaCost: Misclassification Cost-Sensitive Boosting*, 1999.

15.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

15.6.1 Papers

- *The Class Imbalance Problem: A Systematic Study*, 2002.
<https://dl.acm.org/citation.cfm?id=1293954>
- *Types of Cost in Inductive Concept Learning*, 2000.
<https://arxiv.org/abs/cs/0212034>
- *The Foundations Of Cost-sensitive Learning*, 2001.
<https://dl.acm.org/citation.cfm?id=1642224>
- *Cost-Sensitive Learning by Cost-Proportionate Example Weighting*, 2003.
<https://dl.acm.org/citation.cfm?id=952181>
- *Cost-sensitive Learning Methods For Imbalanced Data*, 2010.
<https://ieeexplore.ieee.org/document/5596486>
- *MetaCost: A General Method for Making Classifiers Cost-Sensitive*, 1999.
<https://dl.acm.org/citation.cfm?id=312220>
- *AdaCost: Misclassification Cost-Sensitive Boosting*, 1999.
<https://dl.acm.org/citation.cfm?id=657651>

15.6.2 Books

- *Cost-Sensitive Learning*, *Encyclopedia of Machine Learning*, 2010.
<https://amzn.to/2PamKhX>
- *Cost-Sensitive Machine Learning*, 2011.
<https://amzn.to/2qFgswK>
- Chapter 4 Cost-Sensitive Learning, *Learning from Imbalanced Data Sets*, 2018.
<https://amzn.to/307Xlva>
- *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.
<https://amzn.to/32K9K6d>

15.6.3 Articles

- Confusion matrix, Wikipedia.
https://en.wikipedia.org/wiki/Confusion_matrix

15.7 Summary

In this tutorial, you discovered cost-sensitive learning for imbalanced classification. Specifically, you learned:

- Imbalanced classification problems often value false-positive classification errors differently from false negatives.
- Cost-sensitive learning is a subfield of machine learning that involves explicitly defining and using costs when training machine learning algorithms.
- Cost-sensitive techniques may be divided into three groups, including data sampling, algorithm modifications, and ensemble methods.

15.7.1 Next

In the next tutorial, you will discover how to configure cost-sensitive logistic regression for imbalanced classification.

Chapter 16

Cost-Sensitive Logistic Regression

Logistic regression does not support imbalanced classification directly. Instead, the training algorithm used to fit the logistic regression model must be modified to take the skewed distribution into account. This can be achieved by specifying a class weighting configuration that is used to influence the amount that logistic regression coefficients are updated during training.

The weighting can penalize the model less for errors made on examples from the majority class and penalize the model more for errors made on examples from the minority class. The result is a version of logistic regression that performs better on imbalanced classification tasks, generally referred to as cost-sensitive or weighted logistic regression. In this tutorial, you will discover cost-sensitive logistic regression for imbalanced classification. After completing this tutorial, you will know:

- How standard logistic regression does not support imbalanced classification.
- How logistic regression can be modified to weight model error by class weight when fitting the coefficients.
- How to configure class weight for logistic regression and how to grid search different class weight configurations.

Let's get started.

16.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Imbalanced Classification Dataset
2. Logistic Regression for Imbalanced Classification
3. Weighted Logistic Regression With Scikit-Learn
4. Grid Search Weighted Logistic Regression

16.2 Imbalanced Classification Dataset

Before we dive into the modification of logistic regression for imbalanced classification, let's first define an imbalanced classification dataset. We can use the `make_classification()` function to define a synthetic imbalanced two-class classification dataset. We will generate 10,000 examples with an approximate 1:100 minority to majority class ratio.

```
...
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=2)
```

Listing 16.1: Example of defining an imbalanced binary classification problem.

Once generated, we can summarize the class distribution to confirm that the dataset was created as we expected.

```
...
# summarize class distribution
counter = Counter(y)
print(counter)
```

Listing 16.2: Example of summarizing the class distribution.

Finally, we can create a scatter plot of the examples and color them by class label to help understand the challenge of classifying examples from this dataset.

```
...
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Listing 16.3: Example of creating a scatter plot with dots colored by class label.

Tying this together, the complete example of generating the synthetic dataset and plotting the examples is listed below.

```
# Generate and plot a synthetic imbalanced classification dataset
from collections import Counter
from sklearn.datasets import make_classification
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=2)
# summarize class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Listing 16.4: Example of defining and summarizing an imbalanced classification dataset.

Running the example first creates the dataset and summarizes the class distribution. We can see that the dataset has an approximate 1:100 class distribution with a little less than 10,000 examples in the majority class and 100 in the minority class.

```
Counter({0: 9900, 1: 100})
```

Listing 16.5: Example output from defining and summarizing an imbalanced classification dataset.

Next, a scatter plot of the dataset is created showing the large mass of examples for the majority class (blue) and a small number of examples for the minority class (orange), with some modest class overlap.

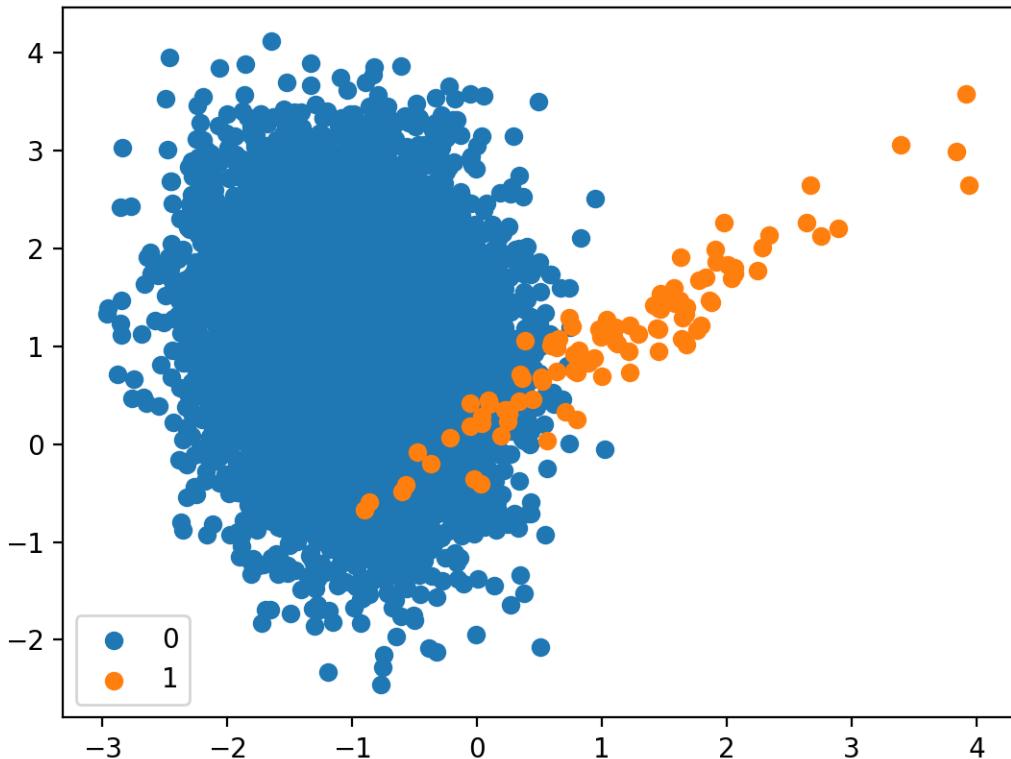


Figure 16.1: Scatter Plot of Binary Classification Dataset With 1 to 100 Class Imbalance.

Next, we can fit a standard logistic regression model on the dataset. We will use repeated cross-validation to evaluate the model, with three repeats of 10-fold cross-validation. The model performance will be reported using the mean ROC area under curve (ROC AUC) averaged over all repeats and folds.

```
...
```

```
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))
```

Listing 16.6: Example of evaluating a model on the dataset.

Tying this together, the complete example of evaluated standard logistic regression on the imbalanced classification problem is listed below.

```
# fit a logistic regression model on an imbalanced classification dataset
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.linear_model import LogisticRegression
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=2)
# define model
model = LogisticRegression(solver='lbfgs')
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))
```

Listing 16.7: Example of evaluating a standard logistic regression algorithm on the imbalanced classification dataset.

Running the example evaluates the standard logistic regression model on the imbalanced dataset and reports the mean ROC AUC.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

We can see that the model has skill, achieving a ROC AUC above 0.5, in this case achieving a mean score of 0.985.

```
Mean ROC AUC: 0.985
```

Listing 16.8: Example output from evaluating a standard logistic regression algorithm on the imbalanced classification dataset.

This provides a baseline for comparison for any modifications performed to the standard logistic regression algorithm.

16.3 Logistic Regression for Imbalanced Classification

Logistic regression is an effective model for binary classification tasks, although by default, it is not effective at imbalanced classification. Logistic regression can be modified to be better

suited for imbalanced classification. The coefficients of the logistic regression algorithm are fit using an optimization algorithm that minimizes the negative log likelihood (loss) for the model on the training dataset.

$$\min \sum_{i=1}^n -(\log(yhat_i) \times y_i + \log(1 - yhat_i) \times (1 - y_i)) \quad (16.1)$$

This involves the repeated use of the model to make predictions followed by an adaptation of the coefficients in a direction that reduces the loss of the model. The calculation of the loss for a given set of coefficients can be modified to take the class balance into account. By default, the errors for each class may be considered to have the same weighting, say 1.0. These weightings can be adjusted based on the importance of each class.

$$\min \sum_{i=1}^n -(w_0 \times \log(yhat_i) \times y_i + w_1 \times \log(1 - yhat_i) \times (1 - y_i)) \quad (16.2)$$

The weighting is applied to the loss so that smaller weight values result in a smaller error value, and in turn, less update to the model coefficients. A larger weight value results in a larger error calculation, and in turn, more update to the model coefficients.

- **Small Weight:** Less importance, less update to the model coefficients.
- **Large Weight:** More importance, more update to the model coefficients.

As such, the modified version of logistic regression is referred to as Weighted Logistic Regression, Class-Weighted Logistic Regression or Cost-Sensitive Logistic Regression. The weightings are sometimes referred to as importance weightings. Although straightforward to implement, the challenge of weighted logistic regression is the choice of the weighting to use for each class.

16.4 Weighted Logistic Regression with Scikit-Learn

The scikit-learn Python machine learning library provides an implementation of logistic regression that supports class weighting. The `LogisticRegression` class provides the `class_weight` argument that can be specified as a model hyperparameter. The `class_weight` is a dictionary that defines each class label (e.g. 0 and 1) and the weighting to apply in the calculation of the negative log likelihood when fitting the model. For example, a 1 to 1 weighting for each class 0 and 1 can be defined as follows:

```
...
# define model
weights = {0:1.0, 1:1.0}
model = LogisticRegression(solver='lbfgs', class_weight=weights)
```

Listing 16.9: Example of defining the default class weighting for logistic regression.

The class weighing can be defined multiple ways; for example:

- **Domain expertise**, determined by talking to subject matter experts.

- **Tuning**, determined by a hyperparameter search such as a grid search.
- **Heuristic**, specified using a general best practice.

A best practice for using the class weighting is to use the inverse of the class distribution present in the training dataset. For example, the class distribution of the training dataset is a 1:100 ratio for the minority class to the majority class. The inversion of this ratio could be used with 1 for the majority class and 100 for the minority class; for example:

```
...
# define model
weights = {0:1.0, 1:100.0}
model = LogisticRegression(solver='lbfgs', class_weight=weights)
```

Listing 16.10: Example of defining the imbalanced weighting for logistic regression as integers.

We might also define the same ratio using fractions and achieve the same result; for example:

```
...
# define model
weights = {0:0.01, 1:1.0}
model = LogisticRegression(solver='lbfgs', class_weight=weights)
```

Listing 16.11: Example of defining the imbalanced weighting for logistic regression as fractions.

We can evaluate the logistic regression algorithm with a class weighting using the same evaluation procedure defined in the previous section. We would expect that the class-weighted version of logistic regression to perform better than the standard version of logistic regression without any class weighting. The complete example is listed below.

```
# weighted logistic regression model on an imbalanced classification dataset
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.linear_model import LogisticRegression
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=2)
# define model
weights = {0:0.01, 1:1.0}
model = LogisticRegression(solver='lbfgs', class_weight=weights)
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))
```

Listing 16.12: Example of evaluating a class-weighted logistic regression algorithm on the imbalanced classification dataset.

Running the example prepares the synthetic imbalanced classification dataset, then evaluates the class-weighted version of logistic regression using repeated cross-validation.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

The mean ROC AUC score is reported, in this case showing a better score than the unweighted version of logistic regression, 0.989 as compared to 0.985.

```
Mean ROC AUC: 0.989
```

Listing 16.13: Example output from evaluating a class-weighted logistic regression algorithm on the imbalanced classification dataset.

The scikit-learn library provides an implementation of the best practice heuristic for the class weighting. It is implemented via the `compute_class_weight()` function and is calculated as:

$$\frac{n_samples}{n_classes \times n_samples_with_class} \quad (16.3)$$

We can test this calculation manually on our dataset. For example, we have 10,000 examples in the dataset, 9900 in class 0, and 100 in class 1. The weighting for class 0 is calculated as:

$$\begin{aligned} \text{weighting} &= \frac{n_samples}{n_classes \times n_samples_with_class} \\ &= \frac{10000}{2 \times 9900} \\ &= \frac{10000}{19800} \\ &= 0.05 \end{aligned} \quad (16.4)$$

The weighting for class 1 is calculated as:

$$\begin{aligned} \text{weighting} &= \frac{n_samples}{n_classes \times n_samples_with_class} \\ &= \frac{10000}{2 \times 100} \\ &= \frac{10000}{200} \\ &= 50 \end{aligned} \quad (16.5)$$

We can confirm these calculations by calling the `compute_class_weight()` function and specifying the `class_weight` as ‘balanced’. For example:

```
# calculate heuristic class weighting
from sklearn.utils.class_weight import compute_class_weight
from sklearn.datasets import make_classification
# generate 2 class dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=2)
# calculate class weighting
weighting = compute_class_weight('balanced', [0,1], y)
print(weighting)
```

Listing 16.14: Example of calculating the class weighting for a dataset.

Running the example, we can see that we can obtain a weighting of about 0.5 for class 0 and a weighting of 50 for class 1.

```
[ 0.50505051 50. ]
```

Listing 16.15: Example output from calculating the class weighting for a dataset.

The values also match our heuristic calculation above for inverting the ratio of the class distribution in the training dataset; for example:

$$0.5 : 50 \equiv 1 : 100 \quad (16.6)$$

We can use the default class balance directly with the `LogisticRegression` class by setting the `class_weight` argument to ‘balanced’. For example:

```
...
# define model
model = LogisticRegression(solver='lbfgs', class_weight='balanced')
```

Listing 16.16: Example of defining the automatic imbalanced weighting for logistic regression.

The complete example is listed below.

```
# weighted logistic regression for class imbalance with heuristic weights
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.linear_model import LogisticRegression
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=2)
# define model
model = LogisticRegression(solver='lbfgs', class_weight='balanced')
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))
```

Listing 16.17: Example of class balanced logistic regression on the imbalanced classification dataset.

Running the example gives the same mean ROC AUC as we achieved by specifying the inverse class ratio manually.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

```
Mean ROC AUC: 0.989
```

Listing 16.18: Example output from class balanced logistic regression on the imbalanced classification dataset.

16.5 Grid Search Weighted Logistic Regression

Using a class weighting that is the inverse ratio of the training data is just a heuristic. It is possible that better performance can be achieved with a different class weighting, and this too will depend on the choice of performance metric used to evaluate the model. In this section, we will grid search a range of different class weightings for weighted logistic regression and discover which results in the best ROC AUC score. We will try the following weightings for class 0 and 1:

- Class 0:100, Class 1:1.
- Class 0:10, Class 1:1.
- Class 0:1, Class 1:1.
- Class 0:1, Class 1:10.
- Class 0:1, Class 1:100.

These can be defined as grid search parameters for the `GridSearchCV` class as follows:

```
...
# define grid
balance = [{0:100,1:1}, {0:10,1:1}, {0:1,1:1}, {0:1,1:10}, {0:1,1:100}]
param_grid = dict(class_weight=balance)
```

Listing 16.19: Example of defining a grid of class weights.

We can perform the grid search on these parameters using repeated cross-validation and estimate model performance using ROC AUC:

```
...
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# define grid search
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=cv,
    scoring='roc_auc')
```

Listing 16.20: Example of performing a grid search of class weights.

Once executed, we can summarize the best configuration as well as all of the results as follows:

```
...
# report the best configuration
print('Best: %f using %s' % (grid_result.best_score_, grid_result.best_params_))
# report all configurations
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print('%f (%f) with: %r' % (mean, stdev, param))
```

Listing 16.21: Example of summarizing the results of a grid search of class weights.

Tying this together, the example below grid searches five different class weights for logistic regression on the imbalanced dataset. We might expect that the heuristic class weighing is the best performing configuration.

```

# grid search class weights with logistic regression for imbalanced classification
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.linear_model import LogisticRegression
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=2)
# define model
model = LogisticRegression(solver='lbfgs')
# define grid
balance = [{0:100,1:1}, {0:10,1:1}, {0:1,1:1}, {0:1,1:10}, {0:1,1:100}]
param_grid = dict(class_weight=balance)
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# define grid search
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=cv,
    scoring='roc_auc')
# execute the grid search
grid_result = grid.fit(X, y)
# report the best configuration
print('Best: %f using %s' % (grid_result.best_score_, grid_result.best_params_))
# report all configurations
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print('%f (%f) with: %r' % (mean, stdev, param))

```

Listing 16.22: Example of a grid search of class weights for logistic regression on the imbalanced classification dataset.

Running the example evaluates each class weighting using repeated k -fold cross-validation and reports the best configuration and the associated mean ROC AUC score.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the 1:100 majority to minority class weighting achieved the best mean ROC score. This matches the configuration for the general heuristic. It might be interesting to explore even more severe class weightings to see their effect on the mean ROC AUC score.

```

Best: 0.989077 using {'class_weight': {0: 1, 1: 100}}
0.982498 (0.016722) with: {'class_weight': {0: 100, 1: 1}}
0.983623 (0.015760) with: {'class_weight': {0: 10, 1: 1}}
0.985387 (0.013890) with: {'class_weight': {0: 1, 1: 1}}
0.988044 (0.010384) with: {'class_weight': {0: 1, 1: 10}}
0.989077 (0.006865) with: {'class_weight': {0: 1, 1: 100}}

```

Listing 16.23: Example output from a grid search of class weights for logistic regression on the imbalanced classification dataset.

16.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

16.6.1 Papers

- *Logistic Regression in Rare Events Data*, 2001.
<https://dash.harvard.edu/handle/1/4125045>
- *The Estimation of Choice Probabilities from Choice Based Samples*, 1977.
<https://www.jstor.org/stable/1914121>

16.6.2 Books

- *Learning from Imbalanced Data Sets*, 2018.
<https://amzn.to/307Xlva>
- *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.
<https://amzn.to/32K9K6d>

16.6.3 APIs

- `sklearn.utils.class_weight.compute_class_weight` API.
https://scikit-learn.org/stable/modules/generated/sklearn.utils.class_weight.compute_class_weight.html
- `sklearn.linear_model.LogisticRegression` API.
https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
- `sklearn.model_selection.GridSearchCV` API.
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

16.7 Summary

In this tutorial, you discovered cost-sensitive logistic regression for imbalanced classification. Specifically, you learned:

- How standard logistic regression does not support imbalanced classification.
- How logistic regression can be modified to weight model error by class weight when fitting the coefficients.
- How to configure class weight for logistic regression and how to grid search different class weight configurations.

16.7.1 Next

In the next tutorial, you will discover how to configure cost-sensitive decision trees for imbalanced classification.

Chapter 17

Cost-Sensitive Decision Trees

The decision tree algorithm is effective for balanced classification, although it does not perform well on imbalanced datasets. The split points of the tree are chosen to best separate examples into two groups with minimum mixing. When both groups are dominated by examples from one class, the criterion used to select a split point will see good separation, when in fact, the examples from the minority class are being ignored.

This problem can be overcome by modifying the criterion used to evaluate split points to take the importance of each class into account, referred to generally as the weighted split-point or weighted decision tree. In this tutorial, you will discover the weighted decision tree for imbalanced classification. After completing this tutorial, you will know:

- How the standard decision tree algorithm does not support imbalanced classification.
- How the decision tree algorithm can be modified to weight model error by class weight when selecting splits.
- How to configure class weight for the decision tree algorithm and how to grid search different class weight configurations.

Let's get started.

17.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Imbalanced Classification Dataset
2. Decision Trees for Imbalanced Classification
3. Weighted Decision Trees With Scikit-Learn
4. Grid Search Weighted Decision Trees

17.2 Imbalanced Classification Dataset

Before we dive into the modification of decision trees for imbalanced classification, let's first define an imbalanced classification dataset. We can use the `make_classification()` function to define a synthetic imbalanced two-class classification dataset. We will generate 10,000 examples with an approximate 1:100 minority to majority class ratio.

```
...
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=3)
```

Listing 17.1: Example of defining an imbalanced binary classification problem.

Once generated, we can summarize the class distribution to confirm that the dataset was created as we expected.

```
...
# summarize class distribution
counter = Counter(y)
print(counter)
```

Listing 17.2: Example of summarizing the class distribution.

Finally, we can create a scatter plot of the examples and color them by class label to help understand the challenge of classifying examples from this dataset.

```
...
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Listing 17.3: Example of creating a scatter plot with dots colored by class label.

Tying this together, the complete example of generating the synthetic dataset and plotting the examples is listed below.

```
# Generate and plot a synthetic imbalanced classification dataset
from collections import Counter
from sklearn.datasets import make_classification
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=3)
# summarize class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Listing 17.4: Example of defining and summarizing an imbalanced classification dataset.

Running the example first creates the dataset and summarizes the class distribution. We can see that the dataset has an approximate 1:100 class distribution with a little less than 10,000 examples in the majority class and 100 in the minority class.

```
Counter({0: 9900, 1: 100})
```

Listing 17.5: Example output from defining and summarizing an imbalanced classification dataset.

Next, a scatter plot of the dataset is created showing the large mass of examples for the majority class (blue) and a small number of examples for the minority class (orange), with some modest class overlap.

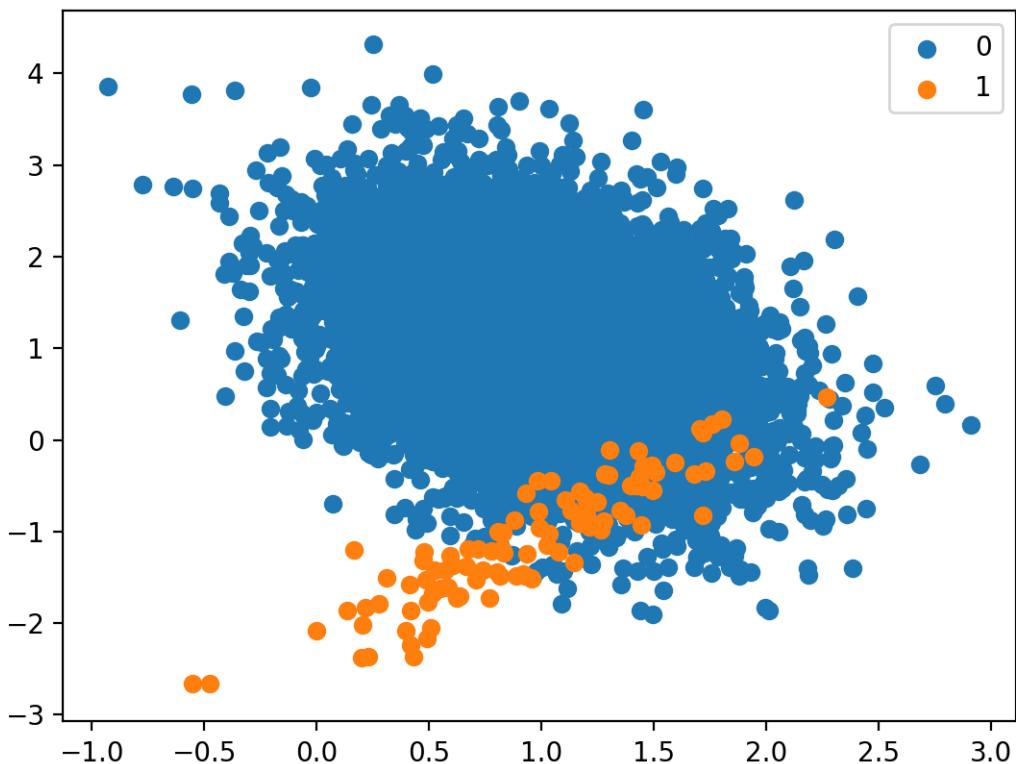


Figure 17.1: Scatter Plot of a Binary Classification Dataset With 1 to 100 Class Imbalance.

Next, we can fit a standard decision tree model on the dataset. A decision tree can be defined using the `DecisionTreeClassifier` class in the scikit-learn library.

```
...
# define model
model = DecisionTreeClassifier()
```

Listing 17.6: Example of defining a standard decision tree algorithm.

We will use repeated cross-validation to evaluate the model, with three repeats of 10-fold cross-validation. The model performance will be reported using the mean ROC area under curve (ROC AUC) averaged over all repeats and folds.

```
...
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))
```

Listing 17.7: Example of evaluating a model on the dataset.

Tying this together, the complete example of defining and evaluating a standard decision tree model on the imbalanced classification problem is listed below. Decision trees are an effective model for binary classification tasks, although by default, they are not effective at imbalanced classification.

```
# fit a decision tree on an imbalanced classification dataset
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.tree import DecisionTreeClassifier
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=3)
# define model
model = DecisionTreeClassifier()
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))
```

Listing 17.8: Example of evaluating a standard decision tree algorithm on the imbalanced classification dataset.

Running the example evaluates the standard decision tree model on the imbalanced dataset and reports the mean ROC AUC.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

We can see that the model has skill, achieving a ROC AUC above 0.5, in this case achieving a mean score of 0.746.

Mean ROC AUC: 0.746

Listing 17.9: Example output from evaluating a standard decision tree algorithm on the imbalanced classification dataset.

This provides a baseline for comparison for any modifications performed to the standard decision tree algorithm.

17.3 Decision Trees for Imbalanced Classification

The decision tree algorithm is also known as Classification and Regression Trees (CART) and involves growing a tree to classify examples from the training dataset. The tree can be thought to divide the training dataset, where examples progress down the decision points of the tree to arrive in the leaves of the tree and are assigned a class label.

The tree is constructed by splitting the training dataset using values for variables in the dataset. At each point, the split in the data that results in the purest (least mixed) groups of examples is chosen in a greedy manner. Here, purity means a clean separation of examples into groups where a group of examples of all 0 or all 1 class is the purest, and a 50-50 mixture of both classes is the least pure. Purity is most commonly calculated using Gini impurity, although it can also be calculated using entropy.

The calculation of a purity measure involves calculating the probability of an example of a given class being misclassified by a split. Calculating these probabilities involves summing the number of examples in each class within each group. The splitting criterion can be updated to not only take the purity of the split into account, but also be weighted by the importance of each class.

Our intuition for cost-sensitive tree induction is to modify the weight of an instance proportional to the cost of misclassifying the class to which the instance belonged ...

— *An Instance-weighting Method To Induce Cost-sensitive Trees*, 2002.

This can be achieved by replacing the count of examples in each group by a weighted sum, where the coefficient is provided to weight the sum. Larger weight is assigned to the class with more importance, and a smaller weight is assigned to a class with less importance.

- **Small Weight:** Less importance, lower impact on node purity.
- **Large Weight:** More importance, higher impact on node purity.

A small weight can be assigned to the majority class, which has the effect of improving (lowering) the purity score of a node that may otherwise look less well sorted. In turn, this may allow more examples from the majority class to be classified for the minority class, better accommodating those examples in the minority class.

Higher weights [are] assigned to instances coming from the class with a higher value of misclassification cost.

— Page 71, *Learning from Imbalanced Data Sets*, 2018.

As such, this modification of the decision tree algorithm is referred to as a weighted decision tree, a class-weighted decision tree, or a cost-sensitive decision tree. Modification of the split point calculation is the most common, although there has been a lot of research into a range of other modifications of the decision tree construction algorithm to better accommodate a class imbalance.

17.4 Weighted Decision Tree With Scikit-Learn

The scikit-learn Python machine learning library provides an implementation of the decision tree algorithm that supports class weighting. The `DecisionTreeClassifier` class provides the `class_weight` argument that can be specified as a model hyperparameter. The `class_weight` is a dictionary that defines each class label (e.g. 0 and 1) and the weighting to apply in the calculation of group purity for splits in the decision tree when fitting the model. For example, a 1 to 1 weighting for each class 0 and 1 can be defined as follows:

```
...
# define model
weights = {0:1.0, 1:1.0}
model = DecisionTreeClassifier(class_weight=weights)
```

Listing 17.10: Example of defining the default class weighting for a decision tree.

The class weighing can be defined multiple ways; for example:

- **Domain expertise**, determined by talking to subject matter experts.
- **Tuning**, determined by a hyperparameter search such as a grid search.
- **Heuristic**, specified using a general best practice.

A best practice for using the class weighting is to use the inverse of the class distribution present in the training dataset. For example, the class distribution of the test dataset is a 1:100 ratio for the minority class to the majority class. The invert of this ratio could be used with 1 for the majority class and 100 for the minority class. For example:

```
...
# define model
weights = {0:1.0, 1:100.0}
model = DecisionTreeClassifier(class_weight=weights)
```

Listing 17.11: Example of defining the imbalanced weighting for a decision tree as integers.

We might also define the same ratio using fractions and achieve the same result. For example:

```
...
# define model
weights = {0:0.01, 1:1.0}
model = DecisionTreeClassifier(class_weight=weights)
```

Listing 17.12: Example of defining the imbalanced weighting for a decision tree as fractions.

This heuristic is available directly by setting the `class_weight` to ‘`balanced`’. For example:

```
...
# define model
model = DecisionTreeClassifier(class_weight='balanced')
```

Listing 17.13: Example of defining the automatic imbalanced weighting for a decision tree.

We can evaluate the decision tree algorithm with a class weighting using the same evaluation procedure defined in the previous section. We would expect the class-weighted version of the decision tree to perform better than the standard version of the decision tree without any class weighting. The complete example is listed below.

```
# decision tree with class weight on an imbalanced classification dataset
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.tree import DecisionTreeClassifier
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=3)
# define model
model = DecisionTreeClassifier(class_weight='balanced')
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))
```

Listing 17.14: Example of evaluating a class weighted decision tree algorithm on the imbalanced classification dataset.

Running the example prepares the synthetic imbalanced classification dataset, then evaluates the class-weighted version of the decision tree algorithm using repeated cross-validation.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

The mean ROC AUC score is reported, in this case, showing a better score than the unweighted version of the decision tree algorithm: 0.759 as compared to 0.746.

Mean ROC AUC: 0.759

Listing 17.15: Example output from evaluating a class weighted decision tree algorithm on the imbalanced classification dataset.

17.5 Grid Search Weighted Decision Tree

Using a class weighting that is the inverse ratio of the training data is just a heuristic. It is possible that better performance can be achieved with a different class weighting, and this too will depend on the choice of performance metric used to evaluate the model. In this section, we will grid search a range of different class weightings for the weighted decision tree and discover which results in the best ROC AUC score. We will try the following weightings for class 0 and 1:

- Class 0:100, Class 1:1.
- Class 0:10, Class 1:1.
- Class 0:1, Class 1:1.
- Class 0:1, Class 1:10.
- Class 0:1, Class 1:100.

These can be defined as grid search parameters for the `GridSearchCV` class as follows:

```
...
# define grid
balance = [{0:100,1:1}, {0:10,1:1}, {0:1,1:1}, {0:1,1:10}, {0:1,1:100}]
param_grid = dict(class_weight=balance)
```

Listing 17.16: Example of defining a grid of class weights.

We can perform the grid search on these parameters using repeated cross-validation and estimate model performance using ROC AUC:

```
...
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# define grid search
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=cv,
scoring='roc_auc')
```

Listing 17.17: Example of performing a grid search of class weights.

Once executed, we can summarize the best configuration as well as all of the results as follows:

```
...
# report the best configuration
print('Best: %f using %s' % (grid_result.best_score_, grid_result.best_params_))
# report all configurations
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print('%f (%f) with: %r' % (mean, stdev, param))
```

Listing 17.18: Example of summarizing the results of a grid search of class weights.

Tying this together, the example below grid searches five different class weights for the decision tree algorithm on the imbalanced dataset. We might expect that the heuristic class weighing is the best performing configuration.

```
# grid search class weights with decision tree for imbalance classification
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.tree import DecisionTreeClassifier
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=3)
# define model
model = DecisionTreeClassifier()
# define grid
balance = [{0:100,1:1}, {0:10,1:1}, {0:1,1:1}, {0:1,1:10}, {0:1,1:100}]
param_grid = dict(class_weight=balance)
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# define grid search
```

```

grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=cv,
    scoring='roc_auc')
# execute the grid search
grid_result = grid.fit(X, y)
# report the best configuration
print('Best: %f using %s' % (grid_result.best_score_, grid_result.best_params_))
# report all configurations
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print('%f (%f) with: %r' % (mean, stdev, param))

```

Listing 17.19: Example of a grid search of class weights for a decision tree on the imbalanced classification dataset.

Running the example evaluates each class weighting using repeated k -fold cross-validation and reports the best configuration and the associated mean ROC AUC score.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the 1:100 majority to minority class weighting achieved the best mean ROC score. This matches the configuration for the general heuristic. It might be interesting to explore even more severe class weightings to see their effect on the mean ROC AUC score.

```

Best: 0.752643 using {'class_weight': {0: 1, 1: 100}}
0.737306 (0.080007) with: {'class_weight': {0: 100, 1: 1}}
0.747306 (0.075298) with: {'class_weight': {0: 10, 1: 1}}
0.740606 (0.074948) with: {'class_weight': {0: 1, 1: 1}}
0.747407 (0.068104) with: {'class_weight': {0: 1, 1: 10}}
0.752643 (0.073195) with: {'class_weight': {0: 1, 1: 100}}

```

Listing 17.20: Example output from a grid search of class weights for a decision tree on the imbalanced classification dataset.

17.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

17.6.1 Papers

- *An Instance-weighting Method To Induce Cost-sensitive Trees*, 2002.
<https://ieeexplore.ieee.org/document/1000348>

17.6.2 Books

- *Learning from Imbalanced Data Sets*, 2018.
<https://amzn.to/307Xlva>
- *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.
<https://amzn.to/32K9K6d>

17.6.3 APIs

- `sklearn.utils.class_weight.compute_class_weight` API.
https://scikit-learn.org/stable/modules/generated/sklearn.utils.class_weight.compute_class_weight.html
- `sklearn.tree.DecisionTreeClassifier` API.
<https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html>
- `sklearn.model_selection.GridSearchCV` API.
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

17.7 Summary

In this tutorial, you discovered the weighted decision tree for imbalanced classification. Specifically, you learned:

- How the standard decision tree algorithm does not support imbalanced classification.
- How the decision tree algorithm can be modified to weight model error by class weight when selecting splits.
- How to configure class weight for the decision tree algorithm and how to grid search different class weight configurations.

17.7.1 Next

In the next tutorial, you will discover how to configure cost-sensitive SVMs for imbalanced classification.

Chapter 18

Cost-Sensitive Support Vector Machines

The Support Vector Machine algorithm is effective for balanced classification, although it does not perform well on imbalanced datasets. The SVM algorithm finds a hyperplane decision boundary that best splits the examples into two classes. The split is made soft through the use of a margin that allows some points to be misclassified. By default, this margin favors the majority class on imbalanced datasets, although it can be updated to take the importance of each class into account and dramatically improve the performance of the algorithm on datasets with skewed class distributions.

This modification of SVM that weighs the margin proportional to the class importance is often referred to as weighted SVM, or cost-sensitive SVM. In this tutorial, you will discover weighted support vector machines for imbalanced classification. After completing this tutorial, you will know:

- How the standard support vector machine algorithm is limited for imbalanced classification.
- How the support vector machine algorithm can be modified to weight the margin penalty proportional to class importance during training.
- How to configure class weight for the SVM and how to grid search different class weight configurations.

Let's get started.

18.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Imbalanced Classification Dataset
2. SVM for Imbalanced Classification
3. Weighted SVM With Scikit-Learn
4. Grid Search Weighted SVM

18.2 Imbalanced Classification Dataset

Before we dive into the modification of SVM for imbalanced classification, let's first define an imbalanced classification dataset. We can use the `make_classification()` function to define a synthetic imbalanced two-class classification dataset. We will generate 10,000 examples with an approximate 1:100 minority to majority class ratio.

```
...
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
```

Listing 18.1: Example of defining an imbalanced binary classification problem.

Once generated, we can summarize the class distribution to confirm that the dataset was created as we expected.

```
...
# summarize class distribution
counter = Counter(y)
print(counter)
```

Listing 18.2: Example of summarizing the class distribution.

Finally, we can create a scatter plot of the examples and color them by class label to help understand the challenge of classifying examples from this dataset.

```
...
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Listing 18.3: Example of creating a scatter plot with dots colored by class label.

Tying this together, the complete example of generating the synthetic dataset and plotting the examples is listed below.

```
# Generate and plot a synthetic imbalanced classification dataset
from collections import Counter
from sklearn.datasets import make_classification
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
# summarize class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Listing 18.4: Example of defining and summarizing an imbalanced classification dataset.

Running the example first creates the dataset and summarizes the class distribution. We can see that the dataset has an approximate 1:100 class distribution with a little less than 10,000 examples in the majority class and 100 in the minority class.

```
Counter({0: 9900, 1: 100})
```

Listing 18.5: Example output from defining and summarizing an imbalanced classification dataset.

Next, a scatter plot of the dataset is created showing the large mass of examples for the majority class (blue) and a small number of examples for the minority class (orange), with some modest class overlap.

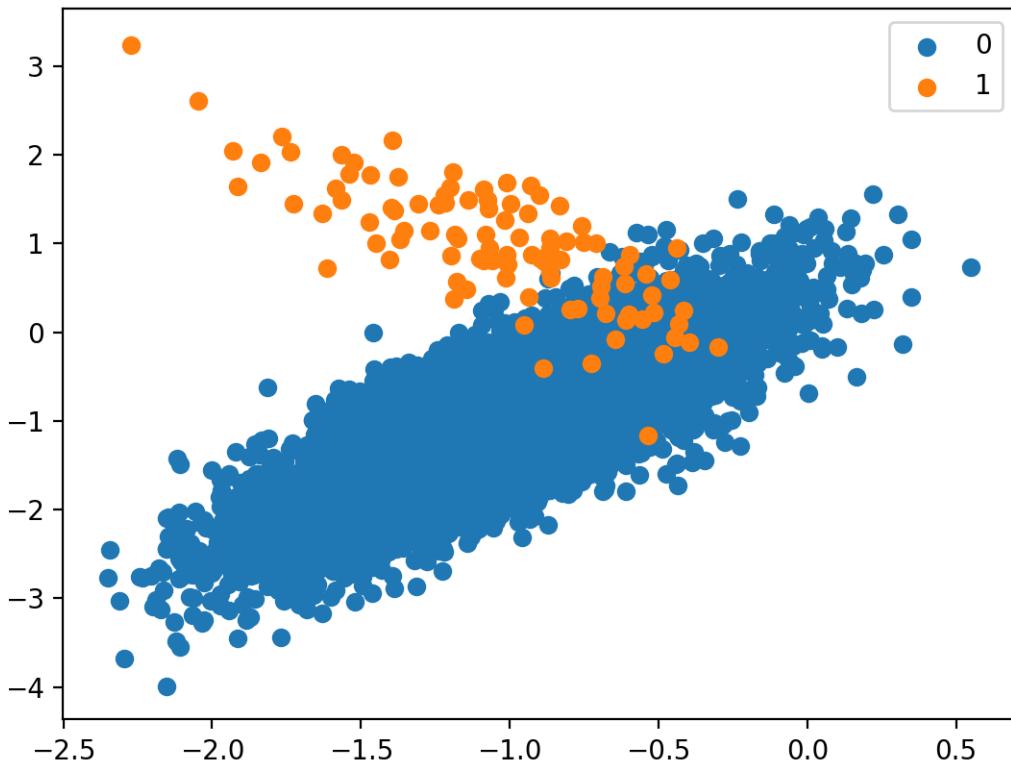


Figure 18.1: Scatter Plot of Binary Classification Dataset With 1 to 100 Class Imbalance.

Next, we can fit a standard SVM model on the dataset. An SVM can be defined using the SVC class in the scikit-learn library.

```
...
# define model
model = SVC(gamma='scale')
```

Listing 18.6: Example of defining a standard SVM algorithm.

We will use repeated cross-validation to evaluate the model, with three repeats of 10-fold cross-validation. The model performance will be reported using the mean ROC area under curve (ROC AUC) averaged over all repeats and folds.

```
...
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))
```

Listing 18.7: Example of evaluating a model on the dataset.

Tying this together, the complete example of defining and evaluating a standard SVM model on the imbalanced classification problem is listed below.

SVMs are effective models for binary classification tasks, although by default, they are not effective at imbalanced classification.

```
# fit a svm on an imbalanced classification dataset
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.svm import SVC
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
# define model
model = SVC(gamma='scale')
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))
```

Listing 18.8: Example of evaluating a standard SVM on the imbalanced classification dataset.

Running the example evaluates the standard SVM model on the imbalanced dataset and reports the mean ROC AUC.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

We can see that the model has skill, achieving a ROC AUC above 0.5, in this case achieving a mean score of 0.804.

```
Mean ROC AUC: 0.804
```

Listing 18.9: Example output from evaluating a standard SVM on the imbalanced classification dataset.

This provides a baseline for comparison for any modifications performed to the standard SVM algorithm.

18.3 SVM for Imbalanced Classification

Support Vector Machines, or SVMs for short, are an effective nonlinear machine learning algorithm. The SVM training algorithm seeks a line or hyperplane that best separates the classes. The hyperplane is defined by a margin that maximizes the distance between the decision boundary and the closest examples from each of the two classes.

Loosely speaking, the margin is the distance between the classification boundary and the closest training set point.

— Pages 343-344, *Applied Predictive Modeling*, 2013.

The data may be transformed using a kernel to allow linear hyperplanes to be defined to separate the classes in a transformed feature space that corresponds to a nonlinear class boundary in the original feature space. Common kernel transforms include a linear, polynomial, and radial basis function transform. This transformation of data is referred to as the *kernel trick*. Typically, the classes are not separable, even with data transforms. As such, the margin is softened to allow some points to appear on the wrong side of the decision boundary. This softening of the margin is controlled by a regularization hyperparameter referred to as the soft-margin parameter, lambda, or capital- C (C).

... where C stands for the regularization parameter that controls the trade-off between maximizing the separation margin between classes and minimizing the number of misclassified instances.

— Page 125, *Learning from Imbalanced Data Sets*, 2018.

A value of $C = 0$ indicates a hard margin and no tolerance for violations of the margin. Small positive values allow some violation, whereas large integer values, such as 1, 10, and 100 allow for a much softer margin.

... [C] determines the number and severity of the violations to the margin (and to the hyperplane) that we will tolerate. We can think of C as a budget for the amount that the margin can be violated by the n observations.

— Page 347, *An Introduction to Statistical Learning: with Applications in R*, 2013.

Although effective, SVMs perform poorly when there is a severe skew in the class distribution. As such, there are many extensions to the algorithm in order to make them more effective on imbalanced datasets.

Although SVMs often produce effective solutions for balanced datasets, they are sensitive to the imbalance in the datasets and produce suboptimal models.

— Page 86, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

The C parameter is used as a penalty during the fit of the model, specifically the finding of the decision boundary. By default, each class has the same weighting, which means that the softness of the margin is symmetrical. Given that there are significantly more examples in the majority class than the minority class, it means that the soft margin and, in turn, the decision boundary will favor the majority class.

... [the] learning algorithm will favor the majority class, as concentrating on it will lead to a better trade-off between classification error and margin maximization. This will come at the expense of minority class, especially when the imbalance ratio is high, as then ignoring the minority class will lead to better optimization results.

— Page 126, *Learning from Imbalanced Data Sets*, 2018.

Perhaps the simplest and most common extension to SVM for imbalanced classification is to weight the C value in proportion to the importance of each class.

To accommodate these factors in SVMs an instance-level weighted modification was proposed. [...] Values of weights may be given depending on the imbalance ratio between classes or individual instance complexity factors.

— Page 130, *Learning from Imbalanced Data Sets*, 2018.

Specifically, each example in the training dataset has its own penalty term (C) used in the calculation for the margin when fitting the SVM model. The value of an example's C -value can be calculated as a weighting of the global C -value, where the weight is defined proportional to the class distribution.

$$C_i = \text{weight}_i \times C \quad (18.1)$$

A larger weighting can be used for the minority class, allowing the margin to be softer, whereas a smaller weighting can be used for the majority class, forcing the margin to be harder and preventing misclassified examples.

- **Small Weight:** Smaller C value, larger penalty for misclassified examples.
- **Larger Weight:** Larger C value, smaller penalty for misclassified examples.

This has the effect of encouraging the margin to contain the majority class with less flexibility, but allow the minority class to be flexible with misclassification of majority class examples onto the minority class side if needed.

That is, the modified SVM algorithm would not tend to skew the separating hyperplane toward the minority class examples to reduce the total misclassifications, as the minority class examples are now assigned with a higher misclassification cost.

— Page 89, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

This modification of SVM may be referred to as Weighted Support Vector Machine, or more generally, Class-Weighted SVM, Instance-Weighted SVM, or Cost-Sensitive SVM.

The basic idea is to assign different weights to different data points such that the WSVM training algorithm learns the decision surface according to the relative importance of data points in the training data set.

— *A Weighted Support Vector Machine For Data Classification*, 2007.

18.4 Weighted SVM With Scikit-Learn

The scikit-learn Python machine learning library provides an implementation of the SVM algorithm that supports class weighting. The `LinearSVC` and `SVC` classes provide the `class_weight` argument that can be specified as a model hyperparameter. The `class_weight` is a dictionary that defines each class label (e.g. 0 and 1) and the weighting to apply to the C value in the calculation of the soft margin. For example, a 1 to 1 weighting for each class 0 and 1 can be defined as follows:

```
...
# define model
weights = {0:1.0, 1:1.0}
model = SVC(gamma='scale', class_weight=weights)
```

Listing 18.10: Example of defining the default class weighting for an SVM.

The class weighing can be defined multiple ways; for example:

- **Domain expertise**, determined by talking to subject matter experts.
- **Tuning**, determined by a hyperparameter search such as a grid search.
- **Heuristic**, specified using a general best practice.

A best practice for using the class weighting is to use the inverse of the class distribution present in the training dataset. For example, the class distribution of the test dataset is a 1:100 ratio for the minority class to the majority class. The inverse of this ratio could be used with 1 for the majority class and 100 for the minority class; for example:

```
...
# define model
weights = {0:1.0, 1:100.0}
model = SVC(gamma='scale', class_weight=weights)
```

Listing 18.11: Example of defining the imbalanced weighting for an SVM as integers.

We might also define the same ratio using fractions and achieve the same result; for example:

```
...
# define model
weights = {0:0.01, 1:1.0}
model = SVC(gamma='scale', class_weight=weights)
```

Listing 18.12: Example of defining the imbalanced weighting for an SVM as fractions.

This heuristic is available directly by setting the `class_weight` to ‘`balanced`’. For example:

```
...
# define model
model = SVC(gamma='scale', class_weight='balanced')
```

Listing 18.13: Example of defining the automatic imbalanced weighting for an SVM.

We can evaluate the SVM algorithm with a class weighting using the same evaluation procedure defined in the previous section. We would expect the class-weighted version of SVM to perform better than the standard version of the SVM without any class weighting. The complete example is listed below.

```
# svm with class weight on an imbalanced classification dataset
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.svm import SVC
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
# define model
model = SVC(gamma='scale', class_weight='balanced')
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))
```

Listing 18.14: Example of evaluating a class weighted SVM on the imbalanced classification dataset.

Running the example prepares the synthetic imbalanced classification dataset, then evaluates the class-weighted version of the SVM algorithm using repeated cross-validation.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

The mean ROC AUC score is reported, in this case, showing a better score than the unweighted version of the SVM algorithm, 0.964 as compared to 0.804.

Mean ROC AUC: 0.964

Listing 18.15: Example output from evaluating a class weighted SVM on the imbalanced classification dataset.

18.5 Grid Search Weighted SVM

Using a class weighting that is the inverse ratio of the distribution observed in the training data is just a heuristic. It is possible that better performance can be achieved with a different class weighting, and this too will depend on the choice of performance metric used to evaluate the model. In this section, we will grid search a range of different class weightings for weighted SVM and discover which results in the best ROC AUC score. We will try the following weightings for class 0 and 1:

- Class 0: 100, Class 1: 1
- Class 0: 10, Class 1: 1
- Class 0: 1, Class 1: 1
- Class 0: 1, Class 1: 10

- Class 0: 1, Class 1: 100

These can be defined as grid search parameters for the `GridSearchCV` class as follows:

```
...
# define grid
balance = [{0:100,1:1}, {0:10,1:1}, {0:1,1:1}, {0:1,1:10}, {0:1,1:100}]
param_grid = dict(class_weight=balance)
```

Listing 18.16: Example of defining a grid of class weights.

We can perform the grid search on these parameters using repeated cross-validation and estimate model performance using ROC AUC:

```
...
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# define grid search
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=cv,
    scoring='roc_auc')
```

Listing 18.17: Example of performing a grid search of class weights.

Once executed, we can summarize the best configuration as well as all of the results as follows:

```
...
# report the best configuration
print('Best: %f using %s' % (grid_result.best_score_, grid_result.best_params_))
# report all configurations
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print('%f (%f) with: %r' % (mean, stdev, param))
```

Listing 18.18: Example of summarizing the results of a grid search of class weights.

Tying this together, the example below grid searches five different class weights for the SVM algorithm on the imbalanced dataset. We might expect that the heuristic class weighing is the best performing configuration.

```
# grid search class weights with svm for imbalance classification
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.svm import SVC
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
# define model
model = SVC(gamma='scale')
# define grid
balance = [{0:100,1:1}, {0:10,1:1}, {0:1,1:1}, {0:1,1:10}, {0:1,1:100}]
param_grid = dict(class_weight=balance)
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
```

```

# define grid search
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=cv,
    scoring='roc_auc')
# execute the grid search
grid_result = grid.fit(X, y)
# report the best configuration
print('Best: %f using %s' % (grid_result.best_score_, grid_result.best_params_))
# report all configurations
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print('%f (%f) with: %r' % (mean, stdev, param))

```

Listing 18.19: Example of a grid search of class weights for an SVM on the imbalanced classification dataset.

Running the example evaluates each class weighting using repeated k -fold cross-validation and reports the best configuration and the associated mean ROC AUC score.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the 1:100 majority to minority class weighting achieved the best mean ROC score. This matches the configuration for the general heuristic. It might be interesting to explore even more severe class weightings to see their effect on the mean ROC AUC score.

```

Best: 0.966189 using {'class_weight': {0: 1, 1: 100}}
0.745249 (0.129002) with: {'class_weight': {0: 100, 1: 1}}
0.748407 (0.128049) with: {'class_weight': {0: 10, 1: 1}}
0.803727 (0.103536) with: {'class_weight': {0: 1, 1: 1}}
0.932620 (0.059869) with: {'class_weight': {0: 1, 1: 10}}
0.966189 (0.036310) with: {'class_weight': {0: 1, 1: 100}}

```

Listing 18.20: Example output from a grid search of class weights for an SVM on the imbalanced classification dataset.

18.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

18.6.1 Papers

- Controlling The Sensitivity Of Support Vector Machines, 1999.
https://seis.bristol.ac.uk/~enicgc/pubs/1999/ijcai_ss.pdf
- Weighted Support Vector Machine For Data Classification, 2005.
<https://ieeexplore.ieee.org/document/1555965>
- A Weighted Support Vector Machine For Data Classification, 2007.
<https://www.worldscientific.com/doi/abs/10.1142/S0218001407005703>

- Cost-Sensitive Support Vector Machines, 2012.
<https://arxiv.org/abs/1212.0975>

18.6.2 Books

- *An Introduction to Statistical Learning: with Applications in R*, 2013.
<https://amzn.to/33VF02Z>
- *Applied Predictive Modeling*, 2013.
<https://amzn.to/2W8wnPS>
- *Learning from Imbalanced Data Sets*, 2018.
<https://amzn.to/307Xlva>
- *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.
<https://amzn.to/32K9K6d>

18.6.3 APIs

- `sklearn.utils.class_weight.compute_class_weight` API.
https://scikit-learn.org/stable/modules/generated/sklearn.utils.class_weight.compute_class_weight.html
- `sklearn.svm.SVC` API.
<https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html>
- `sklearn.svm.LinearSVC` API.
<https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html>
- `sklearn.model_selection.GridSearchCV` API.
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

18.6.4 Articles

- Support-vector machine, Wikipedia.
https://en.wikipedia.org/wiki/Support-vector_machine

18.7 Summary

In this tutorial, you discovered weighted support vector machines for imbalanced classification. Specifically, you learned:

- How the standard support vector machine algorithm is limited for imbalanced classification.
- How the support vector machine algorithm can be modified to weight the margin penalty proportional to class importance during training.
- How to configure class weight for the SVM and how to grid search different class weight configurations.

18.7.1 Next

In the next tutorial, you will discover how to configure cost-sensitive artificial neural networks for imbalanced classification.

Chapter 19

Cost-Sensitive Deep Learning in Keras

Deep learning neural networks are a flexible class of machine learning algorithms that perform well on a wide range of problems. Neural networks are trained using the backpropagation of error algorithm that involves calculating errors made by the model on the training dataset and updating the model weights in proportion to those errors. The limitation of this method of training is that examples from each class are treated the same, which for imbalanced datasets means that the model is adapted a lot more for one class than another.

The backpropagation algorithm can be updated to weigh misclassification errors in proportion to the importance of the class, referred to as weighted neural networks or cost-sensitive neural networks. This has the effect of allowing the model to pay more attention to examples from the minority class than the majority class in datasets with a severely skewed class distribution. In this tutorial, you will discover weighted neural networks for imbalanced classification. After completing this tutorial, you will know:

- How the standard neural network algorithm does not support imbalanced classification.
- How the neural network training algorithm can be modified to weight misclassification errors in proportion to class importance.
- How to configure class weight for neural networks and evaluate the effect on model performance.

Let's get started.

Note: This chapter makes use of the Keras library. See Appendix [B](#) for installation instructions, if needed.

19.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Imbalanced Classification Dataset
2. Neural Network Model in Keras
3. Deep Learning for Imbalanced Classification
4. Weighted Neural Network With Keras

19.2 Imbalanced Classification Dataset

Before we dive into the modification of neural networks for imbalanced classification, let's first define an imbalanced classification dataset. We can use the `make_classification()` function to define a synthetic imbalanced two-class classification dataset. We will generate 10,000 examples with an approximate 1:100 minority to majority class ratio.

```
...
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=2, weights=[0.99], flip_y=0, random_state=4)
```

Listing 19.1: Example of defining an imbalanced binary classification problem.

Once generated, we can summarize the class distribution to confirm that the dataset was created as we expected.

```
...
# summarize class distribution
counter = Counter(y)
print(counter)
```

Listing 19.2: Example of summarizing the class distribution.

Finally, we can create a scatter plot of the examples and color them by class label to help understand the challenge of classifying examples from this dataset.

```
...
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Listing 19.3: Example of creating a scatter plot with dots colored by class label.

Tying this together, the complete example of generating the synthetic dataset and plotting the examples is listed below.

```
# Generate and plot a synthetic imbalanced classification dataset
from collections import Counter
from sklearn.datasets import make_classification
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=2, weights=[0.99], flip_y=0, random_state=4)
# summarize class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Listing 19.4: Example of defining and summarizing an imbalanced classification dataset.

Running the example first creates the dataset and summarizes the class distribution. We can see that the dataset has an approximate 1:100 class distribution with a little less than 10,000 examples in the majority class and 100 in the minority class.

```
Counter({0: 9900, 1: 100})
```

Listing 19.5: Example output from defining and summarizing an imbalanced classification dataset.

Next, a scatter plot of the dataset is created showing the large mass of examples for the majority class (blue) and a small number of examples for the minority class (orange), with some modest class overlap.

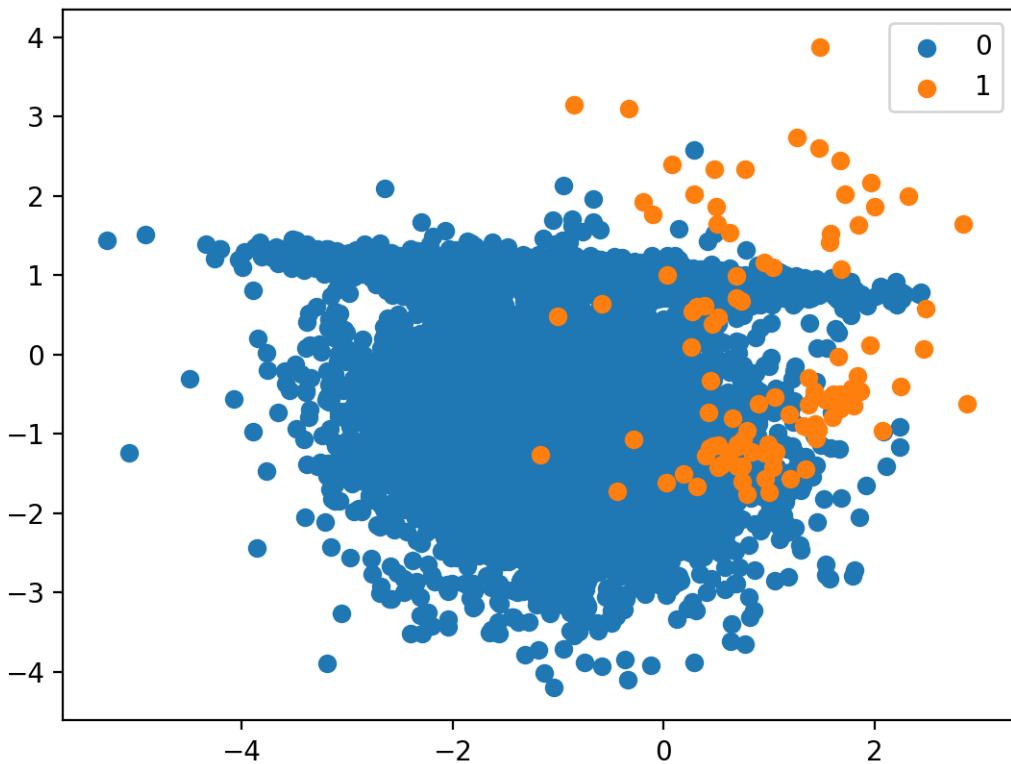


Figure 19.1: Scatter Plot of Binary Classification Dataset With 1 to 100 Class Imbalance.

19.3 Neural Network Model in Keras

Next, we can fit a standard neural network model on the dataset. First, we can define a function to create the synthetic dataset and split it into separate train and test datasets with 5,000 examples in each.

```
# prepare train and test dataset
def prepare_data():
    # generate 2d classification dataset
    X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                               n_clusters_per_class=2, weights=[0.99], flip_y=0, random_state=4)
    # split into train and test
    n_train = 5000
    trainX, testX = X[:n_train, :], X[n_train:, :]
    trainy, testy = y[:n_train], y[n_train:]
    return trainX, trainy, testX, testy
```

Listing 19.6: Example of a function for preparing a dataset for a neural network.

A Multilayer Perceptron neural network can be defined using the Keras deep learning library. We will define a neural network that expects two input variables, has one hidden layer with 10 nodes, then an output layer that predicts the class label. We will use the popular rectified linear (ReLU) activation function in the hidden layer and the sigmoid activation function in the output layer to ensure predictions are probabilities in the range [0,1]. The model will be fit using stochastic gradient descent with the default learning rate and optimized according to cross-entropy loss.

The network architecture and hyperparameters are not optimized to the problem; instead, the network provides a basis for comparison when the training algorithm is later modified to handle the skewed class distribution. The `define_model()` function below defines and returns the model, taking the number of input variables to the network as an argument.

```
# define the neural network model
def define_model(n_input):
    # define model
    model = Sequential()
    # define first hidden layer and visible layer
    model.add(Dense(10, input_dim=n_input, activation='relu',
                   kernel_initializer='he_uniform'))
    # define output layer
    model.add(Dense(1, activation='sigmoid'))
    # define loss and optimizer
    model.compile(loss='binary_crossentropy', optimizer='sgd')
    return model
```

Listing 19.7: Example of a function for defining a neural network model.

Once the model is defined, it can be fit on the training dataset. We will fit the model for 100 training epochs with the default batch size.

```
...
# fit model
model.fit(trainX, trainy, epochs=100, verbose=0)
```

Listing 19.8: Example of fitting the neural network model.

Once fit, we can use the model to make predictions on the test dataset, then evaluate the predictions using the ROC AUC score.

```
...
# make predictions on the test dataset
yhat = model.predict(testX)
```

```
# evaluate the ROC AUC of the predictions
score = roc_auc_score(testy, yhat)
print('ROC AUC: %.3f' % score)
```

Listing 19.9: Example of evaluating the neural network model.

Tying this together, the complete example of fitting a standard neural network model on the imbalanced classification dataset is listed below.

```
# standard neural network on an imbalanced classification dataset
from sklearn.datasets import make_classification
from sklearn.metrics import roc_auc_score
from keras.layers import Dense
from keras.models import Sequential

# prepare train and test dataset
def prepare_data():
    # generate 2d classification dataset
    X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                               n_clusters_per_class=2, weights=[0.99], flip_y=0, random_state=4)
    # split into train and test
    n_train = 5000
    trainX, testX = X[:n_train, :], X[n_train:, :]
    trainy, testy = y[:n_train], y[n_train:]
    return trainX, trainy, testX, testy

# define the neural network model
def define_model(n_input):
    # define model
    model = Sequential()
    # define first hidden layer and visible layer
    model.add(Dense(10, input_dim=n_input, activation='relu',
                    kernel_initializer='he_uniform'))
    # define output layer
    model.add(Dense(1, activation='sigmoid'))
    # define loss and optimizer
    model.compile(loss='binary_crossentropy', optimizer='sgd')
    return model

# prepare dataset
trainX, trainy, testX, testy = prepare_data()
# define the model
n_input = trainX.shape[1]
model = define_model(n_input)
# fit model
model.fit(trainX, trainy, epochs=100, verbose=0)
# make predictions on the test dataset
yhat = model.predict(testX)
# evaluate the ROC AUC of the predictions
score = roc_auc_score(testy, yhat)
print('ROC AUC: %.3f' % score)
```

Listing 19.10: Example of evaluating a standard neural network on the imbalanced classification dataset.

Running the example evaluates the neural network model on the imbalanced dataset and reports the ROC AUC.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, the model achieves a ROC AUC of about 0.949. This suggests that the model has some skill as compared to the naive classifier that has a ROC AUC of 0.5.

```
ROC AUC: 0.949
```

Listing 19.11: Example output from evaluating a standard neural network on the imbalanced classification dataset.

This provides a baseline for comparison for any modifications performed to the standard neural network training algorithm.

19.4 Deep Learning for Imbalanced Classification

Neural network models are commonly trained using the backpropagation of error algorithm. This involves using the current state of the model to make predictions for training set examples, calculating the error for the predictions, then updating the model weights using the error, and assigning credit for the error to different nodes and layers backward from the output layer back through to the input layer. Given the balanced focus on misclassification errors, most standard neural network algorithms are not well suited to datasets with a severely skewed class distribution.

Most of the existing deep learning algorithms do not take the data imbalance problem into consideration. As a result, these algorithms can perform well on the balanced data sets while their performance cannot be guaranteed on imbalanced data sets.

— *Training Deep Neural Networks on Imbalanced Data Sets*, 2016.

This training procedure can be modified so that some examples have more or less error than others.

The misclassification costs can also be taken in account by changing the error function that is being minimized. Instead of minimizing the squared error, the backpropagation learning procedure should minimize the misclassification costs.

— *Cost-Sensitive Learning with Neural Networks*, 1998.

The simplest way to implement this is to use a fixed weighting of error scores for examples based on their class where the prediction error is increased for examples in a more important class and decreased or left unchanged for those examples in a less important class.

... cost sensitive learning methods solve data imbalance problems based on the consideration of the cost associated with misclassifying samples. In particular, it assigns different cost values for the misclassification of the samples.

— *Training Deep Neural Networks on Imbalanced Data Sets*, 2016.

A large error weighting can be applied to those examples in the minority class as they are often more important in an imbalanced classification problem than examples from the majority class.

- **Large Weight:** Assigned to examples from the minority class.
- **Small Weight:** Assigned to examples from the majority class.

This modification to the neural network training algorithm is referred to as a Weighted Neural Network or Cost-Sensitive Neural Network. Typically, careful attention is required when defining the costs or *weightings* to use for cost-sensitive learning. However, for imbalanced classification where only misclassification is the focus, the weighting can use the inverse of the class distribution observed in the training dataset.

19.5 Weighted Neural Network With Keras

The Keras Python deep learning library provides support for class weighting. The `fit()` function that is used to train Keras neural network models takes an argument called `class_weight`. This argument allows you to define a dictionary that maps class integer values to the importance to apply to each class. This function is used to train each different type of neural network, including Multilayer Perceptrons, Convolutional Neural Networks, and Recurrent Neural Networks, therefore the class weighting capability is available to all of those network types. For example, a one-to-one weighting for each class 0 and 1 can be defined as follows:

```
...
# fit model
weights = {0:1, 1:1}
history = model.fit(trainX, trainy, class_weight=weights, ...)
```

Listing 19.12: Example of defining the default class weighting for a neural network.

The class weighing can be defined multiple ways; for example:

- **Domain expertise**, determined by talking to subject matter experts.
- **Tuning**, determined by a hyperparameter search such as a grid search.
- **Heuristic**, specified using a general best practice.

A best practice for using the class weighting is to use the inverse of the class distribution present in the training dataset. For example, the class distribution of the test dataset is a 1:100 ratio for the minority class to the majority class. The invert of this ratio could be used with 1 for the majority class and 100 for the minority class, for example:

```
...
# fit model
weights = {0:1, 1:100}
history = model.fit(trainX, trainy, class_weight=weights, ...)
```

Listing 19.13: Example of defining the imbalanced weighting for neural network as integers.

Fractions that represent the same ratio do not have the same effect. For example, using 0.01 and 0.99 for the majority and minority classes respectively may result in worse performance than using 1 and 100 (it does in this case).

```
...
# fit model
weights = {0:0.01, 1:0.99}
history = model.fit(trainX, trainy, class_weight=weights, ...)
```

Listing 19.14: Example of defining the imbalanced weighting for a neural network as fractions.

The reason is that the error for examples drawn from both the majority class and the minority class is reduced. Further, the reduction in error from the majority class is dramatically scaled down to very small numbers that may have limited or only a very minor effect on model weights. As such integers are recommended to represent the class weightings, such as 1 for no change and 100 for misclassification errors for class 1 having 100-times more impact or penalty than misclassification errors for class 0.

We can evaluate the neural network algorithm with a class weighting using the same evaluation procedure defined in the previous section. We would expect the class-weighted version of the neural network to perform better than the version of the training algorithm without any class weighting. The complete example is listed below.

```
# class weighted neural network on an imbalanced classification dataset
from sklearn.datasets import make_classification
from sklearn.metrics import roc_auc_score
from keras.layers import Dense
from keras.models import Sequential

# prepare train and test dataset
def prepare_data():
    # generate 2d classification dataset
    X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                               n_clusters_per_class=2, weights=[0.99], flip_y=0, random_state=4)
    # split into train and test
    n_train = 5000
    trainX, testX = X[:n_train, :], X[n_train:, :]
    trainy, testy = y[:n_train], y[n_train:]
    return trainX, trainy, testX, testy

# define the neural network model
def define_model(n_input):
    # define model
    model = Sequential()
    # define first hidden layer and visible layer
    model.add(Dense(10, input_dim=n_input, activation='relu',
                    kernel_initializer='he_uniform'))
    # define output layer
    model.add(Dense(1, activation='sigmoid'))
    # define loss and optimizer
    model.compile(loss='binary_crossentropy', optimizer='sgd')
    return model

# prepare dataset
trainX, trainy, testX, testy = prepare_data()
# get the model
```

```
n_input = trainX.shape[1]
model = define_model(n_input)
# fit model
weights = {0:1, 1:100}
history = model.fit(trainX, trainy, class_weight=weights, epochs=100, verbose=0)
# evaluate model
yhat = model.predict(testX)
score = roc_auc_score(testy, yhat)
print('ROC AUC: %.3f' % score)
```

Listing 19.15: Example of evaluating a class weighted neural network on the imbalanced classification dataset.

Running the example prepares the synthetic imbalanced classification dataset, then evaluates the class-weighted version of the neural network training algorithm.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

The ROC AUC score is reported, in this case showing a better score than the unweighted version of the training algorithm, or about 0.973 as compared to about 0.949.

```
ROC AUC: 0.973
```

Listing 19.16: Example output from evaluating a class weighted neural network on the imbalanced classification dataset.

19.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

19.6.1 Papers

- *Cost-Sensitive Learning with Neural Networks*, 1998.
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.13.8285>
- *Training Deep Neural Networks on Imbalanced Data Sets*, 2016.
<https://ieeexplore.ieee.org/abstract/document/7727770>

19.6.2 Books

- *Learning from Imbalanced Data Sets*, 2018.
<https://amzn.to/307Xlva>
- *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.
<https://amzn.to/32K9K6d>

19.6.3 APIs

- `sklearn.datasets.make_classification` API.

https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification.html

- Keras Model API.

<https://keras.io/models/sequential/>

19.7 Summary

In this tutorial, you discovered weighted neural networks for imbalanced classification. Specifically, you learned:

- How the standard neural network algorithm does not support imbalanced classification.
- How the neural network training algorithm can be modified to weight misclassification errors in proportion to class importance.
- How to configure class weight for neural networks and evaluate the effect on model performance.

19.7.1 Next

In the next tutorial, you will discover how to configure cost-sensitive XGBoost models for imbalanced classification.

Chapter 20

Cost-Sensitive Gradient Boosting with XGBoost

The XGBoost algorithm is effective for a wide range of regression and classification predictive modeling problems. It is an efficient implementation of the stochastic gradient boosting algorithm and offers a range of hyperparameters that give fine-grained control over the model training procedure. Although the algorithm performs well in general, even on imbalanced classification datasets, it offers a way to tune the training algorithm to pay more attention to misclassification of the minority class for datasets with a skewed class distribution.

This modified version of XGBoost is referred to as Class Weighted XGBoost or Cost-Sensitive XGBoost and can offer better performance on binary classification problems with a severe class imbalance. In this tutorial, you will discover weighted XGBoost for imbalanced classification. After completing this tutorial, you will know:

- How gradient boosting works from a high level and how to develop an XGBoost model for classification.
- How the XGBoost training algorithm can be modified to weight error gradients proportional to positive class importance during training.
- How to configure the positive class weight for the XGBoost training algorithm and how to grid search different configurations.

Let's get started.

Note: This chapter makes use of the XGBoost library. See Appendix B for installation instructions, if needed.

20.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Imbalanced Classification Dataset
2. XGBoost Model for Classification
3. Weighted XGBoost for Class Imbalance
4. Tune the Class Weighting Hyperparameter

20.2 Imbalanced Classification Dataset

Before we dive into XGBoost for imbalanced classification, let's first define an imbalanced classification dataset. We can use the `make_classification()` scikit-learn function to define a synthetic imbalanced two-class classification dataset. We will generate 10,000 examples with an approximate 1:100 minority to majority class ratio.

```
...
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=2, weights=[0.99], flip_y=0, random_state=7)
```

Listing 20.1: Example of defining an imbalanced binary classification problem.

Once generated, we can summarize the class distribution to confirm that the dataset was created as we expected.

```
...
# summarize class distribution
counter = Counter(y)
print(counter)
```

Listing 20.2: Example of summarizing the class distribution.

Finally, we can create a scatter plot of the examples and color them by class label to help understand the challenge of classifying examples from this dataset.

```
...
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Listing 20.3: Example of creating a scatter plot with dots colored by class label.

Tying this together, the complete example of generating the synthetic dataset and plotting the examples is listed below.

```
# Generate and plot a synthetic imbalanced classification dataset
from collections import Counter
from sklearn.datasets import make_classification
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=2, weights=[0.99], flip_y=0, random_state=7)
# summarize class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Listing 20.4: Example of defining and summarizing an imbalanced classification dataset.

Running the example first creates the dataset and summarizes the class distribution. We can see that the dataset has an approximate 1:100 class distribution with a little less than 10,000 examples in the majority class and 100 in the minority class.

```
Counter({0: 9900, 1: 100})
```

Listing 20.5: Example output from defining and summarizing an imbalanced classification dataset.

Next, a scatter plot of the dataset is created showing the large mass of examples for the majority class (blue) and a small number of examples for the minority class (orange), with some modest class overlap.

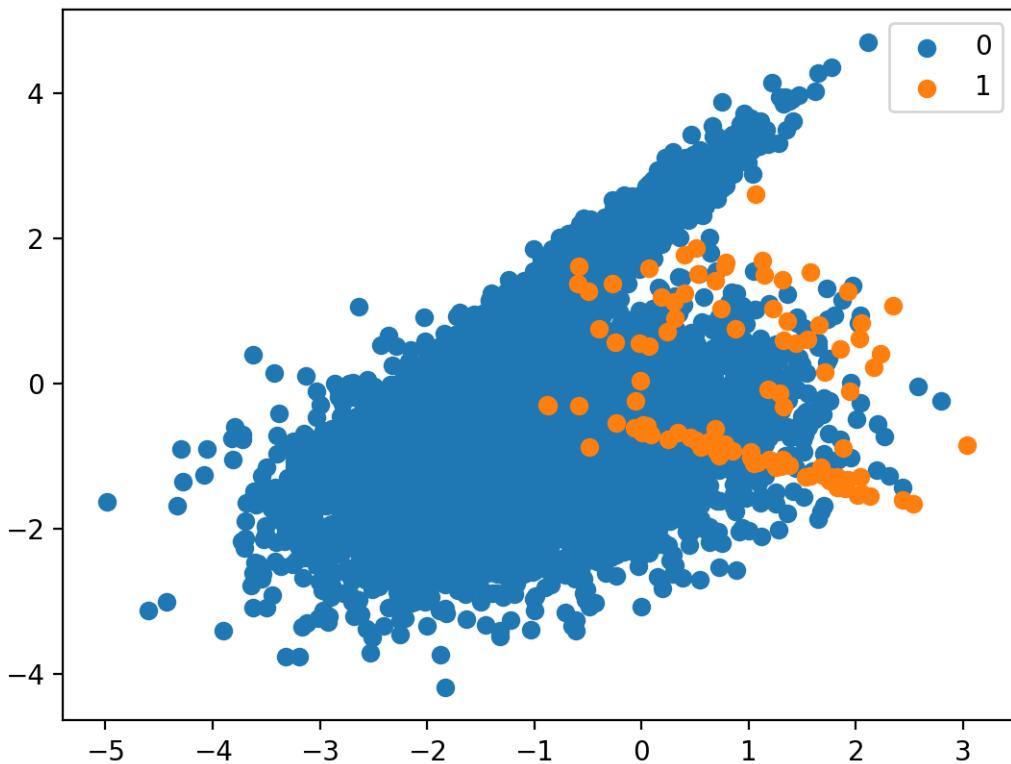


Figure 20.1: Scatter Plot of Binary Classification Dataset With 1 to 100 Class Imbalance.

20.3 XGBoost Model for Classification

XGBoost is short for Extreme Gradient Boosting and is an efficient implementation of the stochastic gradient boosting machine learning algorithm. The stochastic gradient boosting algorithm, also called gradient boosting machines or tree boosting, is a powerful machine learning

technique that performs well or even best on a wide range of challenging machine learning problems.

Tree boosting has been shown to give state-of-the-art results on many standard classification benchmarks.

— *XGBoost: A Scalable Tree Boosting System*, 2016.

It is an ensemble of decision trees algorithm where new trees fix errors of those trees that are already part of the model. Trees are added until no further improvements can be made to the model. XGBoost provides a highly efficient implementation of the stochastic gradient boosting algorithm and access to a suite of model hyperparameters designed to provide control over the model training process.

The most important factor behind the success of XGBoost is its scalability in all scenarios. The system runs more than ten times faster than existing popular solutions on a single machine and scales to billions of examples in distributed or memory-limited settings.

— *XGBoost: A Scalable Tree Boosting System*, 2016.

XGBoost is an effective machine learning model, even on datasets where the class distribution is skewed. Before any modification or tuning is made to the XGBoost algorithm for imbalanced classification, it is important to test the default XGBoost model and establish a baseline in performance. Although the XGBoost library has its own Python API, we can use XGBoost models with the scikit-learn API via the `XGBClassifier` wrapper class. An instance of the model can be instantiated and used just like any other scikit-learn class for model evaluation. For example:

```
...
# define model
model = XGBClassifier()
```

Listing 20.6: Example of defining a standard XGBoost model.

We will use repeated cross-validation to evaluate the model, with three repeats of 10-fold cross-validation. The model performance will be reported using the mean ROC area under curve (ROC AUC) averaged over repeats and all folds.

```
...
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.5f' % mean(scores))
```

Listing 20.7: Example of evaluating a model on the dataset.

Tying this together, the complete example of defining and evaluating a default XGBoost model on the imbalanced classification problem is listed below.

```
# fit xgboost on an imbalanced classification dataset
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from xgboost import XGBClassifier
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=2, weights=[0.99], flip_y=0, random_state=7)
# define model
model = XGBClassifier()
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.5f' % mean(scores))
```

Listing 20.8: Example of evaluating a standard XGBoost on the imbalanced classification dataset.

Running the example evaluates the default XGBoost model on the imbalanced dataset and reports the mean ROC AUC.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

We can see that the model has skill, achieving a ROC AUC above 0.5, in this case achieving a mean score of about 0.957.

Mean ROC AUC: 0.95724

Listing 20.9: Example output from evaluating a standard XGBoost on the imbalanced classification dataset.

This provides a baseline for comparison for any hyperparameter tuning performed for the default XGBoost algorithm.

20.4 Weighted XGBoost for Class Imbalance

Although the XGBoost algorithm performs well for a wide range of challenging problems, it offers a large number of hyperparameters, many of which require tuning in order to get the most out of the algorithm on a given dataset. The implementation provides a hyperparameter designed to tune the behavior of the algorithm for imbalanced classification problems; this is the `scale_pos_weight` hyperparameter.

By default, the `scale_pos_weight` hyperparameter is set to the value of 1.0 and has the effect of weighing the balance of positive examples, relative to negative examples when boosting decision trees. For an imbalanced binary classification dataset, the negative class refers to the majority class (class 0) and the positive class refers to the minority class (class 1).

XGBoost is trained to minimize a loss function and the *gradient* in gradient boosting refers to the steepness of this loss function, e.g. the amount of error. A small gradient means a small

error and, in turn, a small change to the model to correct the error. A large error gradient during training in turn results in a large correction.

- **Small Gradient:** Small error or correction to the model.
- **Large Gradient:** Large error or correction to the model.

Gradients are used as the basis for fitting subsequent trees added to boost or correct errors made by the existing state of the ensemble of decision trees. The `scale_pos_weight` value is used to scale the gradient for the positive class. This has the effect of scaling errors made by the model during training on the positive class and encourages the model to over-correct them. In turn, this can help the model achieve better performance when making predictions on the positive class. Pushed too far, it may result in the model overfitting the positive class at the cost of worse performance on the negative class or both classes.

As such, the `scale_pos_weight` can be used to train a class-weighted or cost-sensitive version of XGBoost for imbalanced classification. A sensible default value to set for the `scale_pos_weight` hyperparameter is the inverse of the class distribution. For example, for a dataset with a 1 to 100 ratio for examples in the minority to majority classes, the `scale_pos_weight` can be set to 100. This will give classification errors made by the model on the minority class (positive class) 100 times more impact, and in turn, 100 times more correction than errors made on the majority class. For example:

```
...
# define model
model = XGBClassifier(scale_pos_weight=100)
```

Listing 20.10: Example of defining an XGBoost with weight scaling.

The XGBoost documentation suggests a fast way to estimate this value using the training dataset, calculated as the total number of examples in the majority class divided by the total number of examples in the minority class.

$$\text{scale_pos_weight} = \frac{\text{total_negative_examples}}{\text{total_positive_examples}} \quad (20.1)$$

For example, we can calculate this value for our synthetic classification dataset. We would expect this to be about 100, or more precisely, 99 given the weighting we used to define the dataset.

```
...
# count examples in each class
counter = Counter(y)
# estimate scale_pos_weight value
estimate = counter[0] / counter[1]
print('Estimate: %.3f' % estimate)
```

Listing 20.11: Example of calculating the positive class scaling weight.

The complete example of estimating the value for the `scale_pos_weight` XGBoost hyperparameter is listed below.

```
# estimate a value for the scale_pos_weight xgboost hyperparameter
from sklearn.datasets import make_classification
from collections import Counter
```

```
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=2, weights=[0.99], flip_y=0, random_state=7)
# count examples in each class
counter = Counter(y)
# estimate scale_pos_weight value
estimate = counter[0] / counter[1]
print('Estimate: %.3f' % estimate)
```

Listing 20.12: Example of estimating the positive weight scale for the imbalanced dataset.

Running the example creates the dataset and estimates the values of the `scale_pos_weight` hyperparameter as 99, as we expected.

```
Estimate: 99.000
```

Listing 20.13: Example output from estimating the positive weight scale for the imbalanced dataset.

We will use this value directly in the configuration of the XGBoost model and evaluate its performance on the dataset using repeated k -fold cross-validation. We would expect some improvement in ROC AUC, although this is not guaranteed depending on the difficulty of the dataset and the chosen configuration of the XGBoost model. The complete example is listed below.

```
# fit balanced xgboost on an imbalanced classification dataset
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from xgboost import XGBClassifier
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=2, weights=[0.99], flip_y=0, random_state=7)
# define model
model = XGBClassifier(scale_pos_weight=99)
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.5f' % mean(scores))
```

Listing 20.14: Example of evaluating a class weighted XGBoost on the imbalanced classification dataset.

Running the example prepares the synthetic imbalanced classification dataset, then evaluates the class-weighted version of the XGBoost training algorithm using repeated cross-validation.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see a modest lift in performance from a ROC AUC of about 0.95724 with `scale_pos_weight=1` in the previous section to a value of 0.95990 with `scale_pos_weight=99`.

```
Mean ROC AUC: 0.95990
```

Listing 20.15: Example output from evaluating a class weighted XGBoost on the imbalanced classification dataset.

20.5 Tune the Class Weighting Hyperparameter

The heuristic for setting the `scale_pos_weight` is effective for many situations. Nevertheless, it is possible that better performance can be achieved with a different class weighting, and this too will depend on the choice of performance metric used to evaluate the model. In this section, we will grid search a range of different class weightings for class-weighted XGBoost and discover which results in the best ROC AUC score. We will try the following weightings for the positive class:

- 1 (*default*)
- 10
- 25
- 50
- 75
- 99 (*recommended*)
- 100
- 1000

These can be defined as grid search parameters for the `GridSearchCV` class as follows:

```
...
# define grid
weights = [1, 10, 25, 50, 75, 99, 100, 1000]
param_grid = dict(scale_pos_weight=weights)
```

Listing 20.16: Example of defining a grid of positive class weights.

We can perform the grid search on these parameters using repeated cross-validation and estimate model performance using ROC AUC:

```
...
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# define grid search
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=cv,
scoring='roc_auc')
```

Listing 20.17: Example of performing a grid search of class weights.

Once executed, we can summarize the best configuration as well as all of the results as follows:

```

...
# report the best configuration
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
# report all configurations
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))

```

Listing 20.18: Example of summarizing the results of a grid search of class weights.

Tying this together, the example below grid searches eight different positive class weights for the XGBoost algorithm on the imbalanced dataset. We might expect that the heuristic class weighing is the best performing configuration.

```

# grid search positive class weights with xgboost for imbalance classification
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RepeatedStratifiedKFold
from xgboost import XGBClassifier
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=2, weights=[0.99], flip_y=0, random_state=7)
# define model
model = XGBClassifier()
# define grid
weights = [1, 10, 25, 50, 75, 99, 100, 1000]
param_grid = dict(scale_pos_weight=weights)
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# define grid search
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=cv,
    scoring='roc_auc')
# execute the grid search
grid_result = grid.fit(X, y)
# report the best configuration
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
# report all configurations
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))

```

Listing 20.19: Example of a grid search of positive class weights for an XGBoost on the imbalanced classification dataset.

Running the example evaluates each positive class weighting using repeated k -fold cross-validation and reports the best configuration and the associated mean ROC AUC score.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the `scale_pos_weight=99` positive class weighting achieved the best mean ROC score. This matches the configuration for the general heuristic. It's interesting to note that almost all values larger than the default value of 1 have a better mean ROC AUC, even the aggressive value of 1,000. It's also interesting to note that a value of 99 performed better from the value of 100, which I may have used if I did not calculate the heuristic as suggested in the XGBoost documentation.

```
Best: 0.959901 using {'scale_pos_weight': 99}
0.957239 (0.031619) with: {'scale_pos_weight': 1}
0.958219 (0.027315) with: {'scale_pos_weight': 10}
0.958278 (0.027438) with: {'scale_pos_weight': 25}
0.959199 (0.026171) with: {'scale_pos_weight': 50}
0.959204 (0.025842) with: {'scale_pos_weight': 75}
0.959901 (0.025499) with: {'scale_pos_weight': 99}
0.959141 (0.025409) with: {'scale_pos_weight': 100}
0.958761 (0.024757) with: {'scale_pos_weight': 1000}
```

Listing 20.20: Example output from a grid search of positive class weights for an XGBoost on the imbalanced classification dataset.

20.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

20.6.1 Papers

- *XGBoost: A Scalable Tree Boosting System*, 2016.
<https://arxiv.org/abs/1603.02754>

20.6.2 Books

- *Learning from Imbalanced Data Sets*, 2018.
<https://amzn.to/307Xlva>
- *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.
<https://amzn.to/32K9K6d>

20.6.3 APIs

- `sklearn.datasets.make_classification` API.
https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification.html
- `xgboost.XGBClassifier` API.
https://xgboost.readthedocs.io/en/latest/python/python_api.html
- XGBoost Parameters, API Documentation.
<https://xgboost.readthedocs.io/en/latest/parameter.html>

- Notes on Parameter Tuning, API Documentation.
https://xgboost.readthedocs.io/en/latest/tutorials/param_tuning.html

20.7 Summary

In this tutorial, you discovered weighted XGBoost for imbalanced classification. Specifically, you learned:

- How gradient boosting works from a high level and how to develop an XGBoost model for classification.
- How the XGBoost training algorithm can be modified to weight error gradients proportional to positive class importance during training.
- How to configure the positive class weight for the XGBoost training algorithm and how to grid search different configurations.

20.7.1 Next

This was the final tutorial in this Part. In the next Part, you will discover advanced techniques that you can use for imbalanced classification.

Part VI

Advanced Algorithms

Chapter 21

Probability Threshold Moving

Classification predictive modeling typically involves predicting a class label. Nevertheless, many machine learning algorithms are capable of predicting a probability or scoring of class membership, and this must be interpreted before it can be mapped to a crisp class label. This is achieved by using a threshold, such as 0.5, where all values equal or greater than the threshold are mapped to one class and all other values are mapped to another class.

For those classification problems that have a severe class imbalance, the default threshold can result in poor performance. As such, a simple and straightforward approach to improving the performance of a classifier that predicts probabilities on an imbalanced classification problem is to tune the threshold used to map probabilities to class labels.

In some cases, such as when using ROC Curves and Precision-Recall Curves, the best or optimal threshold for the classifier can be calculated directly. In other cases, it is possible to use a grid search to tune the threshold and locate the optimal value. In this tutorial, you will discover how to tune the optimal threshold when converting probabilities to crisp class labels for imbalanced classification. After completing this tutorial, you will know:

- The default threshold for interpreting probabilities to class labels is 0.5, and tuning this hyperparameter is called threshold moving.
- How to calculate the optimal threshold for the ROC Curve and Precision-Recall Curve directly.
- How to manually search threshold values for a chosen model and model evaluation metric.

Let's get started.

21.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Converting Probabilities to Class Labels
2. Threshold-Moving for Imbalanced Classification
3. Optimal Threshold for ROC Curve
4. Optimal Threshold for Precision-Recall Curve
5. Optimal Threshold Tuning

21.2 Converting Probabilities to Class Labels

Many machine learning algorithms are capable of predicting a probability or a scoring of class membership. This is useful generally as it provides a measure of the certainty or uncertainty of a prediction. It also provides additional granularity over just predicting the class label that can be interpreted. Some classification tasks require a crisp class label prediction. This means that even though a probability or scoring of class membership is predicted, it must be converted into a crisp class label.

The decision for converting a predicted probability or scoring into a class label is governed by a parameter referred to as the *decision threshold*, *discrimination threshold*, or simply the *threshold*. The default value for the threshold is 0.5 for normalized predicted probabilities or scores in the range between 0 or 1. For example, on a binary classification problem with class labels 0 and 1, normalized predicted probabilities and a threshold of 0.5, then values less than the threshold of 0.5 are assigned to class 0 and values greater than or equal to 0.5 are assigned to class 1.

$$\begin{aligned} \text{Prediction} < 0.5 &= \text{Class0} \\ \text{Prediction} \geq 0.5 &= \text{Class1} \end{aligned} \tag{21.1}$$

The problem is that the default threshold may not represent an optimal interpretation of the predicted probabilities. This might be the case for a number of reasons, such as:

- The predicted probabilities are not calibrated, e.g. those predicted by an SVM or decision tree.
- The metric used to train the model is different from the metric used to evaluate a final model.
- The class distribution is severely skewed.
- The cost of one type of misclassification is more important than another type of misclassification.

Worse still, some or all of these reasons may occur at the same time, such as the use of a neural network model with uncalibrated predicted probabilities on an imbalanced classification problem. As such, there is often the need to change the default decision threshold when interpreting the predictions of a model.

... almost all classifiers generate positive or negative predictions by applying a threshold to a score. The choice of this threshold will have an impact in the trade-offs of positive and negative errors.

— Page 53, *Learning from Imbalanced Data Sets*, 2018.

21.3 Threshold-Moving for Imbalanced Classification

There are many techniques that may be used to address an imbalanced classification problem, such as sampling the training dataset and developing customized versions of machine learning

algorithms. Nevertheless, perhaps the simplest approach to handle a severe class imbalance is to change the decision threshold. Although simple and very effective, this technique is often overlooked by practitioners and research academics alike as was noted by Foster Provost in his 2000 article titled *Machine Learning from Imbalanced Data Sets*.

The bottom line is that when studying problems with imbalanced data, using the classifiers produced by standard machine learning algorithms without adjusting the output threshold may well be a critical mistake.

— *Machine Learning from Imbalanced Data Sets 101*, 2000.

There are many reasons to choose an alternative to the default decision threshold. For example, you may use ROC curves to analyze the predicted probabilities of a model and ROC AUC scores to compare and select a model, although you require crisp class labels from your model. How do you choose the threshold on the ROC Curve that results in the best balance between the true positive rate and the false positive rate?

Alternately, you may use precision-recall curves to analyze the predicted probabilities of a model, precision-recall AUC to compare and select models, and require crisp class labels as predictions. How do you choose the threshold on the Precision-Recall Curve that results in the best balance between precision and recall? You may use a probability-based metric to train, evaluate, and compare models like log loss (cross-entropy) but require crisp class labels to be predicted. How do you choose the optimal threshold from predicted probabilities more generally?

Finally, you may have different costs associated with false positive and false negative misclassification, a so-called cost matrix, but wish to use and evaluate cost-insensitive models and later evaluate their predictions use a cost-sensitive measure. How do you choose a threshold that finds the best trade-off for predictions using the cost matrix?

Popular way of training a cost-sensitive classifier without a known cost matrix is to put emphasis on modifying the classification outputs when predictions are being made on new data. This is usually done by setting a threshold on the positive class, below which the negative one is being predicted. The value of this threshold is optimized using a validation set and thus the cost matrix can be learned from training data.

— Page 67, *Learning from Imbalanced Data Sets*, 2018.

The answer to these questions is to search a range of threshold values in order to find the best threshold. In some cases, the optimal threshold can be calculated directly. Tuning or shifting the decision threshold in order to accommodate the broader requirements of the classification problem is generally referred to as *threshold-moving*, *threshold-tuning*, or simply *thresholding*.

It has been stated that trying other methods, such as sampling, without trying by simply setting the threshold may be misleading. The threshold-moving method uses the original training set to train [a model] and then moves the decision threshold such that the minority class examples are easier to be predicted correctly.

— Pages 72, *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

The process involves first fitting the model on a training dataset and making predictions on a test dataset. The predictions are in the form of normalized probabilities, or scores that are transformed into normalized probabilities. Different threshold values are then tried and the resulting crisp labels are evaluated using a chosen evaluation metric. The threshold that achieves the best evaluation metric is then adopted for the model when making predictions on new data in the future. We can summarize this procedure below.

1. Fit Model on the Training Dataset.
2. Predict Probabilities on the Test Dataset.
3. For each threshold in Thresholds:
 - (a) Convert probabilities to Class Labels using the threshold.
 - (b) Evaluate Class Labels.
 - (c) If Score is Better than Best Score.
 - Adopt Threshold.
4. Use Adopted Threshold When Making Class Predictions on New Data.

Although simple, there are a few different approaches to implementing threshold-moving depending on your circumstance. We will take a look at some of the most common examples in the following sections.

21.4 Optimal Threshold for ROC Curve

A ROC curve is a diagnostic plot that evaluates a set of probability predictions made by a model on a test dataset. A set of different thresholds are used to interpret the true positive rate and the false positive rate of the predictions on the positive (minority) class, and the scores are plotted in a line of increasing thresholds to create a curve. The false-positive rate is plotted on the x-axis and the true positive rate is plotted on the y-axis and the plot is referred to as the Receiver Operating Characteristic curve, or ROC curve. A diagonal line on the plot from the bottom-left to top-right indicates the *curve* for a no-skill classifier (predicts the majority class in all cases), and a point in the top left of the plot indicates a model with perfect skill.

The curve is useful to understand the trade-off in the true-positive rate and false-positive rate for different thresholds. The area under the ROC Curve, so-called ROC AUC, provides a single number to summarize the performance of a model in terms of its ROC Curve with a value between 0.5 (no-skill) and 1.0 (perfect skill). The ROC Curve is a useful diagnostic tool for understanding the trade-off for different thresholds and the ROC AUC provides a useful number for comparing models based on their general capabilities.

If crisp class labels are required from a model under such an analysis, then an optimal threshold is required. This would be a threshold on the curve that is closest to the top-left of the plot. Thankfully, there are principled ways of locating this point. First, let's fit a model and calculate a ROC Curve. We can use the `make_classification()` function to create a synthetic binary classification problem with 10,000 examples (rows), 99 percent of which belong to the majority class and 1 percent belong to the minority class.

```
...
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
```

Listing 21.1: Example of defining an imbalanced binary classification problem.

We can then split the dataset using the `train_test_split()` function and use half for the training set and half for the test set.

```
...  
# split into train/test sets  
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2,  
    stratify=y)
```

Listing 21.2: Example of splitting the dataset into train and test sets.

We can then fit a `LogisticRegression` model and use it to make probability predictions on the test set and keep only the probability predictions for the minority class.

```
...
# fit a model
model = LogisticRegression(solver='lbfgs')
model.fit(trainX, trainy)
# predict probabilities
lr_probs = model.predict_proba(testX)
# keep probabilities for the positive outcome only
lr_probs = lr_probs[:, 1]
```

Listing 21.3: Example of fitting a logistic regression model and making predictions.

We can then use the `roc_auc_score()` function to calculate the true-positive rate and false-positive rate for the predictions using a set of thresholds that can then be used to create a ROC Curve plot.

```
...  
# calculate scores  
lr_auc = roc_auc_score(testy, lr_probs)
```

Listing 21.4: Example of evaluating predicted probabilities.

We can tie this all together, defining the dataset, fitting the model, and creating the ROC Curve plot. The complete example is listed below.

```
# roc curve for logistic regression model
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve
from matplotlib import pyplot
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2,
    stratify=y)
# fit a model
```

```
model = LogisticRegression(solver='lbfgs')
model.fit(trainX, trainy)
# predict probabilities
yhat = model.predict_proba(testX)
# keep probabilities for the positive outcome only
yhat = yhat[:, 1]
# calculate roc curves
fpr, tpr, thresholds = roc_curve(testy, yhat)
# plot the roc curve for the model
pyplot.plot([0,1], [0,1], linestyle='--', label='No Skill')
pyplot.plot(fpr, tpr, marker='.', label='Logistic')
# axis labels
pyplot.xlabel('False Positive Rate')
pyplot.ylabel('True Positive Rate')
pyplot.legend()
# show the plot
pyplot.show()
```

Listing 21.5: Example of evaluating the predicted probabilities for a logistic regression model using a ROC curve.

Running the example fits a logistic regression model on the training dataset then evaluates it using a range of thresholds on the test set, creating the ROC Curve. We can see that there are a number of points or thresholds close to the top-left of the plot. Which is the threshold that is optimal?

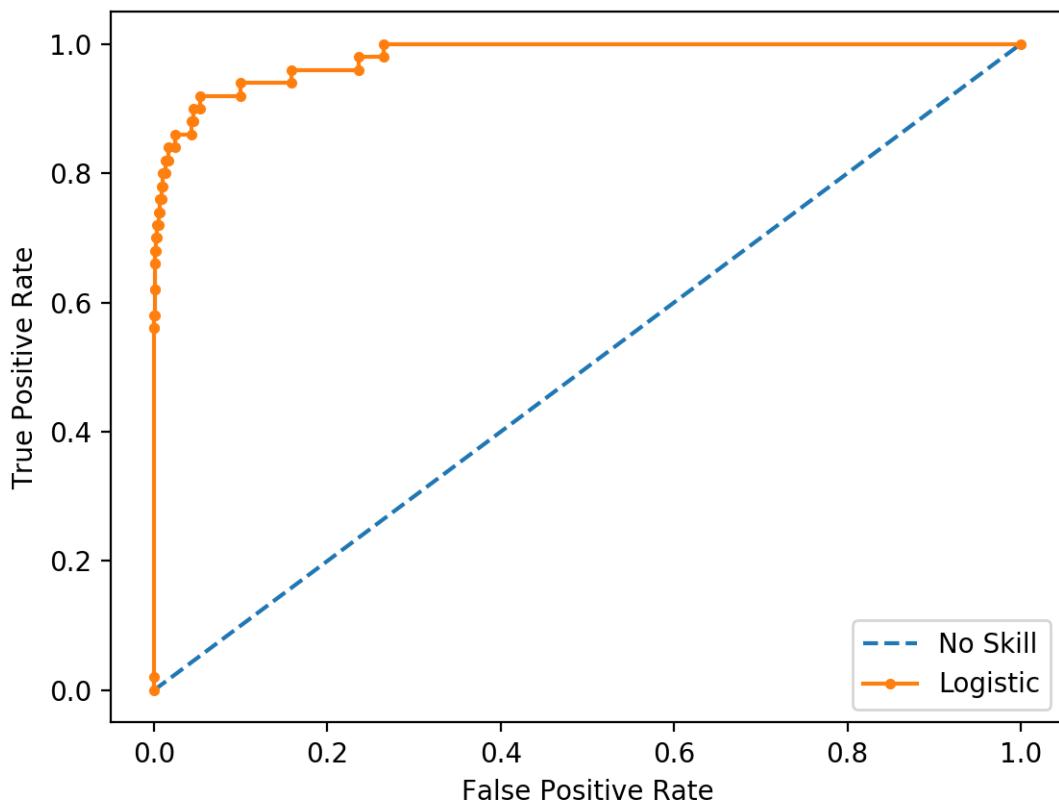


Figure 21.1: ROC Curve Line Plot for Logistic Regression Model for Imbalanced Classification.

There are many ways we could locate the threshold with the optimal balance between false positive and true positive rates. Firstly, the true positive rate is called the Sensitivity. One minus the false-positive rate is called the Specificity. The Geometric Mean or G-mean is a metric for imbalanced classification that, if optimized, will seek a balance between the sensitivity and the specificity.

$$\text{G-mean} = \sqrt{\text{Sensitivity} \times \text{Specificity}} \quad (21.2)$$

One approach would be to test the model with each threshold returned from the call `roc_auc_score()` and select the threshold with the largest G-mean value. Given that we have already calculated the Sensitivity (TPR) and the complement to the Specificity when we calculated the ROC Curve, we can calculate the G-mean for each threshold directly.

```
...
# calculate the g-mean for each threshold
gmeans = sqrt(tpr * (1-fpr))
```

Listing 21.6: Example of calculating the G-means.

Once calculated, we can locate the index for the largest G-mean score and use that index to determine which threshold value to use.

```
...
```

```
# locate the index of the largest g-mean
ix = argmax(gmeans)
print('Best Threshold=%f, G-mean=%.3f' % (thresholds[ix], gmeans[ix]))
```

Listing 21.7: Example of selecting the optimal threshold.

We can also re-draw the ROC Curve and highlight this point. The complete example is listed below.

```
# roc curve for logistic regression model with optimal threshold
from numpy import sqrt
from numpy import argmax
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve
from matplotlib import pyplot
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2,
    stratify=y)
# fit a model
model = LogisticRegression(solver='lbfgs')
model.fit(trainX, trainy)
# predict probabilities
yhat = model.predict_proba(testX)
# keep probabilities for the positive outcome only
yhat = yhat[:, 1]
# calculate roc curves
fpr, tpr, thresholds = roc_curve(testy, yhat)
# calculate the g-mean for each threshold
gmeans = sqrt(tpr * (1-fpr))
# locate the index of the largest g-mean
ix = argmax(gmeans)
print('Best Threshold=%f, G-mean=%.3f' % (thresholds[ix], gmeans[ix]))
# plot the roc curve for the model
pyplot.plot([0,1], [0,1], linestyle='--', label='No Skill')
pyplot.plot(fpr, tpr, marker='.', label='Logistic')
pyplot.scatter(fpr[ix], tpr[ix], marker='o', color='black', label='Best')
# axis labels
pyplot.xlabel('False Positive Rate')
pyplot.ylabel('True Positive Rate')
pyplot.legend()
# show the plot
pyplot.show()
```

Listing 21.8: Example of locating the optimal probability threshold for a logistic regression model.

Running the example first locates the optimal threshold and reports this threshold and the G-mean score. In this case, we can see that the optimal threshold is about 0.016153.

```
Best Threshold=0.016153, G-mean=0.933
```

Listing 21.9: Example output from locating the optimal probability threshold for a logistic

regression model.

The threshold is then used to locate the true and false positive rates, then this point is drawn on the ROC Curve. We can see that the point for the optimal threshold is a large black dot and it appears to be closest to the top-left of the plot.

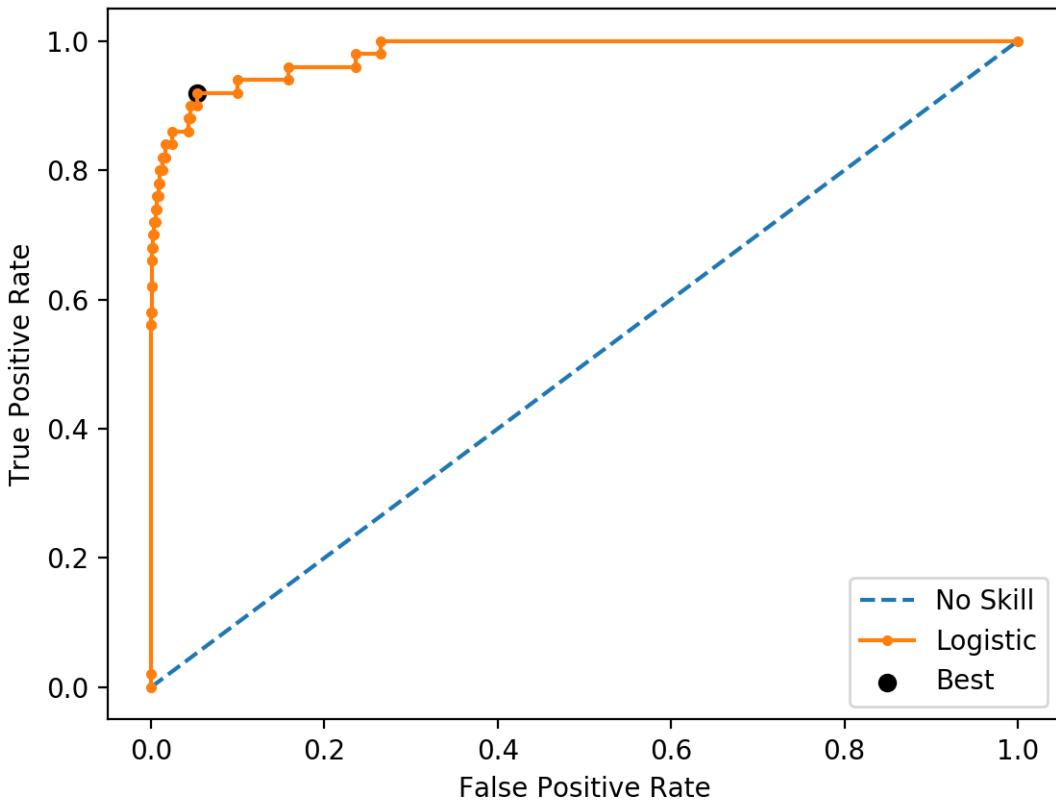


Figure 21.2: ROC Curve Line Plot for Logistic Regression Model for Imbalanced Classification With the Optimal Threshold.

It turns out there is a much faster way to get the same result, called the Youden's J statistic. The statistic is calculated as:

$$J = \text{Sensitivity} + \text{Specificity} - 1 \quad (21.3)$$

Given that we have Sensitivity (TPR) and the complement of the specificity (FPR), we can calculate it as:

$$J = \text{Sensitivity} + (1 - \text{FalsePositiveRate}) - 1 \quad (21.4)$$

Which we can restate as:

$$J = \text{TruePositiveRate} - \text{FalsePositiveRate} \quad (21.5)$$

We can then choose the threshold with the largest J statistic value. For example:

```

...
# calculate roc curves
fpr, tpr, thresholds = roc_curve(testy, yhat)
# get the best threshold
J = tpr - fpr
ix = argmax(J)
best_thresh = thresholds[ix]
print('Best Threshold=%f' % (best_thresh))

```

Listing 21.10: Example of selecting the optimal threshold using the J-statistic.

Plugging this in, the complete example is listed below.

```

# roc curve for logistic regression model with optimal threshold
from numpy import argmax
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2,
    stratify=y)
# fit a model
model = LogisticRegression(solver='lbfgs')
model.fit(trainX, trainy)
# predict probabilities
yhat = model.predict_proba(testX)
# keep probabilities for the positive outcome only
yhat = yhat[:, 1]
# calculate roc curves
fpr, tpr, thresholds = roc_curve(testy, yhat)
# get the best threshold
J = tpr - fpr
ix = argmax(J)
best_thresh = thresholds[ix]
print('Best Threshold=%f' % (best_thresh))

```

Listing 21.11: Example of locating the optimal probability threshold using the J-statistic.

We can see that this simpler approach calculates the optimal statistic directly.

```
Best Threshold=0.016153
```

Listing 21.12: Example output from locating the optimal probability threshold using the J-statistic.

21.5 Optimal Threshold for Precision-Recall Curve

Unlike the ROC Curve, a precision-recall curve focuses on the performance of a classifier on the positive (minority class) only. Precision is the ratio of the number of true positives divided by the sum of the true positives and false positives. It describes how good a model is at predicting

the positive class. Recall is calculated as the ratio of the number of true positives divided by the sum of the true positives and the false negatives. Recall is the same as sensitivity.

A precision-recall curve is calculated by creating crisp class labels for probability predictions across a set of thresholds and calculating the precision and recall for each threshold. A line plot is created for the thresholds in ascending order with recall on the x -axis and precision on the y -axis. A no-skill model is represented by a horizontal line with a precision that is the ratio of positive examples in the dataset (e.g. $\frac{TP}{TP+TN}$), or 0.01 on our synthetic dataset. A perfect skill classifier has full precision and recall with a dot in the top-right corner.

We can use the same model and dataset from the previous section and evaluate the probability predictions for a logistic regression model using a precision-recall curve. The `precision_recall_curve()` function can be used to calculate the curve, returning the precision and recall scores for each threshold as well as the thresholds used.

```
...
# calculate pr-curve
precision, recall, thresholds = precision_recall_curve(testy, yhat)
```

Listing 21.13: Example of calculating the precision-recall curve.

Tying this together, the complete example of calculating a precision-recall curve for a logistic regression on an imbalanced classification problem is listed below.

```
# pr curve for logistic regression model
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_recall_curve
from matplotlib import pyplot
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2,
    stratify=y)
# fit a model
model = LogisticRegression(solver='lbfgs')
model.fit(trainX, trainy)
# predict probabilities
yhat = model.predict_proba(testX)
# keep probabilities for the positive outcome only
yhat = yhat[:, 1]
# calculate pr-curve
precision, recall, thresholds = precision_recall_curve(testy, yhat)
# plot the roc curve for the model
no_skill = len(testy[testy==1]) / len(testy)
pyplot.plot([0,1], [no_skill,no_skill], linestyle='--', label='No Skill')
pyplot.plot(recall, precision, marker='.', label='Logistic')
# axis labels
pyplot.xlabel('Recall')
pyplot.ylabel('Precision')
pyplot.legend()
# show the plot
pyplot.show()
```

Listing 21.14: Example of evaluating the predicted probabilities for a logistic regression model

using a precision-recall curve.

Running the example calculates the precision and recall for each threshold and creates a precision-recall plot showing that the model has some skill across a range of thresholds on this dataset. If we required crisp class labels from this model, which threshold would achieve the best result?

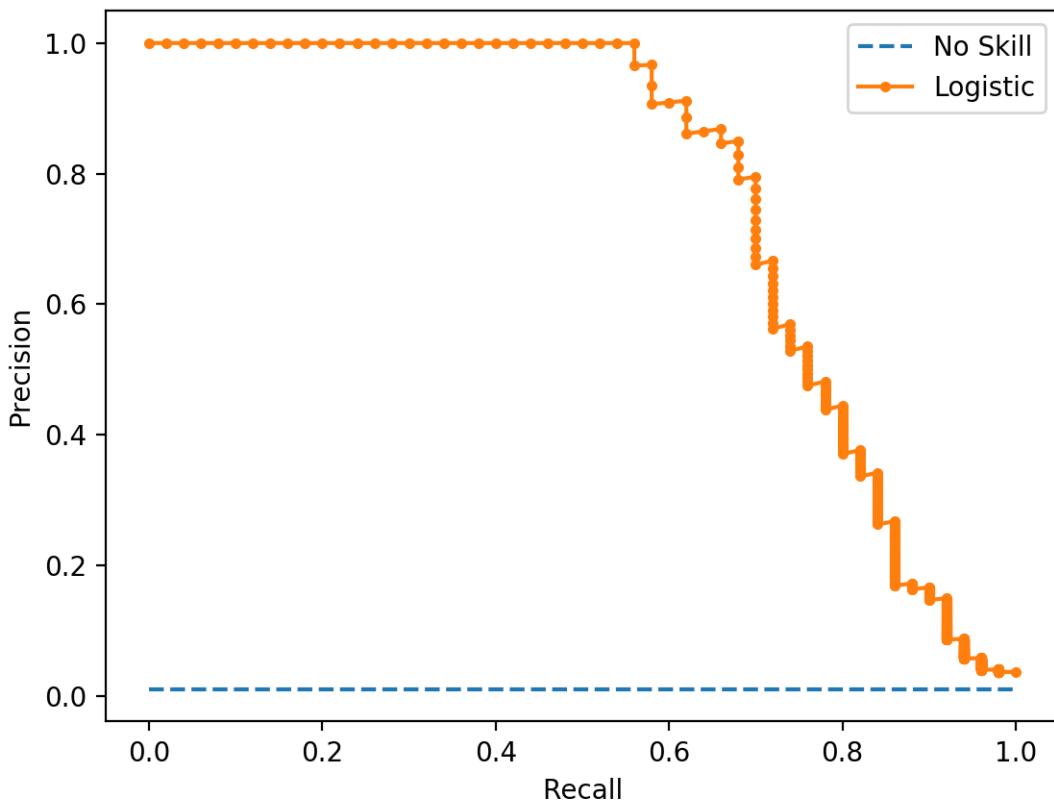


Figure 21.3: Precision-Recall Curve Line Plot for Logistic Regression Model for Imbalanced Classification.

If we are interested in a threshold that results in the best balance of precision and recall, then this is the same as optimizing the F-measure that summarizes the harmonic mean of both measures.

$$\text{F-measure} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad (21.6)$$

As in the previous section, the naive approach to finding the optimal threshold would be to calculate the F-measure for each threshold. We can achieve the same effect by converting the precision and recall measures to F-measure directly; for example:

```
...
# convert to f-measure
fscore = (2 * precision * recall) / (precision + recall)
# locate the index of the largest f-measure
```

```
ix = argmax(fscore)
print('Best Threshold=%f, F-measure=%.3f' % (thresholds[ix], fscore[ix]))
```

Listing 21.15: Example of calculating optimal threshold on the precision-recall curve.

We can then plot the point on the precision-recall curve. The complete example is listed below.

```
# optimal threshold for precision-recall curve with logistic regression model
from numpy import argmax
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_recall_curve
from matplotlib import pyplot
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2,
                                               stratify=y)
# fit a model
model = LogisticRegression(solver='lbfgs')
model.fit(trainX, trainy)
# predict probabilities
yhat = model.predict_proba(testX)
# keep probabilities for the positive outcome only
yhat = yhat[:, 1]
# calculate roc curves
precision, recall, thresholds = precision_recall_curve(testy, yhat)
# convert to f-measure
fscore = (2 * precision * recall) / (precision + recall)
# locate the index of the largest f-measure
ix = argmax(fscore)
print('Best Threshold=%f, F-measure=%.3f' % (thresholds[ix], fscore[ix]))
# plot the roc curve for the model
no_skill = len(testy[testy==1]) / len(testy)
pyplot.plot([0,1], [no_skill,no_skill], linestyle='--', label='No Skill')
pyplot.plot(recall, precision, marker='.', label='Logistic')
pyplot.scatter(recall[ix], precision[ix], marker='o', color='black', label='Best')
# axis labels
pyplot.xlabel('Recall')
pyplot.ylabel('Precision')
pyplot.legend()
# show the plot
pyplot.show()
```

Listing 21.16: Example of calculating the optimal threshold on the precision-recall curve.

Running the example first calculates the F-measure for each threshold, then locates the score and threshold with the largest value. In this case, we can see that the best F-measure was 0.756 achieved with a threshold of about 0.26.

```
Best Threshold=0.256036, F-measure=0.756
```

Listing 21.17: Example output from calculating the optimal threshold on the precision-recall curve.

The precision-recall curve is plotted, and this time the threshold with the optimal F-measure is plotted with a larger black dot. This threshold could then be used when making probability predictions in the future that must be converted from probabilities to crisp class labels.

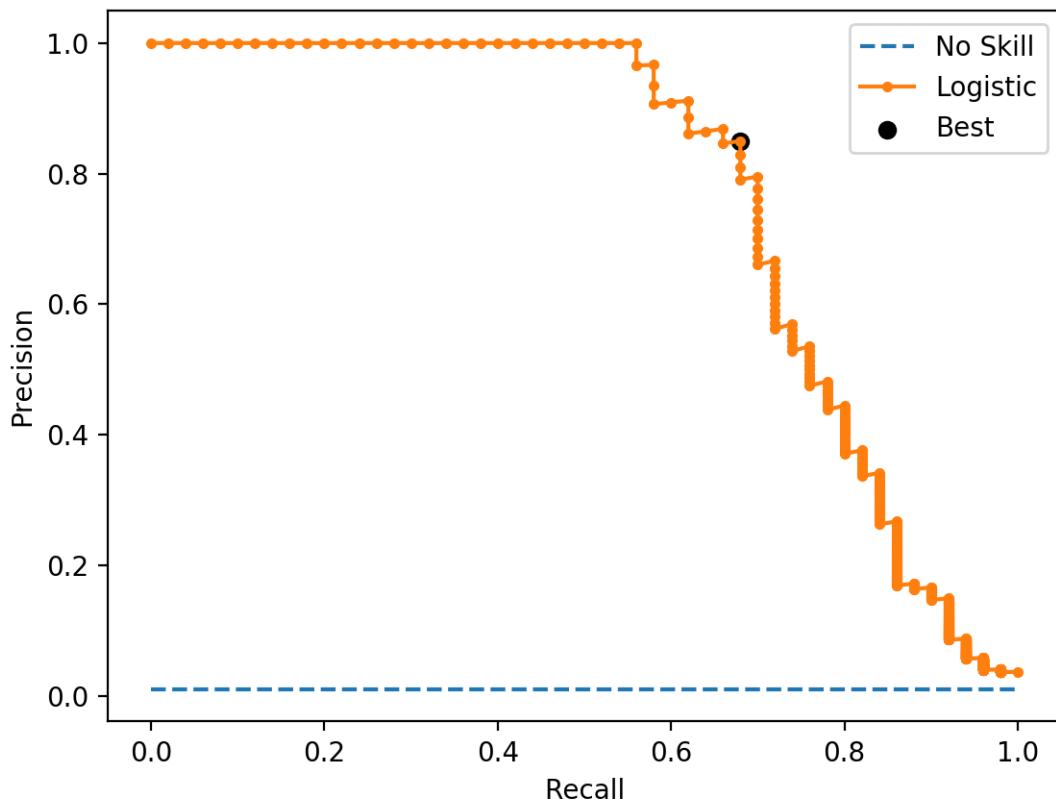


Figure 21.4: Precision-Recall Curve Line Plot for Logistic Regression Model With Optimal Threshold.

21.6 Optimal Threshold Tuning

Sometimes, we simply have a model and we wish to know the best threshold directly. In this case, we can define a set of thresholds and then evaluate predicted probabilities under each in order to find and select the optimal threshold. We can demonstrate this with a worked example.

First, we can fit a logistic regression model on our synthetic classification problem, then predict class labels and evaluate them using the F-measure, which is the harmonic mean of precision and recall. This will use the default threshold of 0.5 when interpreting the probabilities predicted by the logistic regression model. The complete example is listed below.

```
# logistic regression for imbalanced classification
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import f1_score
# generate dataset
```

```

X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2,
                                               stratify=y)
# fit a model
model = LogisticRegression(solver='lbfgs')
model.fit(trainX, trainy)
# predict labels
yhat = model.predict(testX)
# evaluate the model
score = f1_score(testy, yhat)
print('F-measure: %.5f' % score)

```

Listing 21.18: Example of evaluating the F-measure of a logistic regression using the default threshold.

Running the example, we can see that the model achieved an F-measure of about 0.70 on the test dataset.

```
F-measure: 0.70130
```

Listing 21.19: Example output from evaluating the F-measure of a logistic regression using the default threshold.

Now we can use the same model on the same dataset and instead of predicting class labels directly, we can predict probabilities.

```

...
# predict probabilities
yhat = model.predict_proba(testX)

```

Listing 21.20: Example of predicting probabilities.

We only require the probabilities for the positive class.

```

...
# keep probabilities for the positive outcome only
probs = yhat[:, 1]

```

Listing 21.21: Example of retrieving the probabilities for the positive class.

Next, we can then define a set of thresholds to evaluate the probabilities. In this case, we will test all thresholds between 0.0 and 1.0 with a step size of 0.001, that is, we will test 0.0, 0.001, 0.002, 0.003, and so on to 0.999.

```

...
# define thresholds
thresholds = arange(0, 1, 0.001)

```

Listing 21.22: Example of defining the range of thresholds to test.

Next, we need a way of using a single threshold to interpret the predicted probabilities. This can be achieved by mapping all values equal to or greater than the threshold to 1 and all values less than the threshold to 0. We will define a `to_labels()` function to do this that will take the probabilities and threshold as an argument and return an array of integers in {0, 1}.

```
# apply threshold to positive probabilities to create labels
def to_labels(pos_probs, threshold):
    return (pos_probs >= threshold).astype('int')
```

Listing 21.23: Example of a function for applying a threshold to predicted probabilities.

We can then call this function for each threshold and evaluate the resulting labels using the `f1_score()`. We can do this in a single line, as follows:

```
...
# evaluate each threshold
scores = [f1_score(testy, to_labels(probs, t)) for t in thresholds]
```

Listing 21.24: Example of evaluating predicted probabilities using each threshold.

We now have an array of scores that evaluate each threshold in our array of thresholds. All we need to do now is locate the array index that has the largest score (best F-measure) and we will have the optimal threshold and its evaluation.

```
...
# get best threshold
ix = argmax(scores)
print('Threshold=%.3f, F-measure=%.5f' % (thresholds[ix], scores[ix]))
```

Listing 21.25: Example reporting the optimal threshold.

Tying this all together, the complete example of tuning the threshold for the logistic regression model on the synthetic imbalanced classification dataset is listed below.

```
# search thresholds for imbalanced classification
from numpy import arange
from numpy import argmax
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import f1_score

# apply threshold to positive probabilities to create labels
def to_labels(pos_probs, threshold):
    return (pos_probs >= threshold).astype('int')

# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2,
                                                stratify=y)
# fit a model
model = LogisticRegression(solver='lbfgs')
model.fit(trainX, trainy)
# predict probabilities
yhat = model.predict_proba(testX)
# keep probabilities for the positive outcome only
probs = yhat[:, 1]
# define thresholds
thresholds = arange(0, 1, 0.001)
# evaluate each threshold
```

```

scores = [f1_score(testy, to_labels(probs, t)) for t in thresholds]
# get best threshold
ix = argmax(scores)
print('Threshold=%.3f, F-measure=%.5f' % (thresholds[ix], scores[ix]))

```

Listing 21.26: Example of grid searching probability thresholds.

Running the example reports the optimal threshold as 0.251 (compared to the default of 0.5) that achieves an F-measure of about 0.75 (compared to 0.70). You can use this example as a template when tuning the threshold on your own problem, allowing you to substitute your own model, metric, and even resolution of thresholds that you want to evaluate.

```
Threshold=0.251, F-measure=0.75556
```

Listing 21.27: Example output from grid searching probability thresholds.

21.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

21.7.1 Papers

- *Machine Learning from Imbalanced Data Sets 101*, 2000.
<https://www.aaai.org/Library/Workshops/2000/ws00-05-001.php>
- *Training Cost-sensitive Neural Networks With Methods Addressing The Class Imbalance Problem*, 2005.
<https://ieeexplore.ieee.org/document/1549828>

21.7.2 Books

- *Learning from Imbalanced Data Sets*, 2018.
<https://amzn.to/307Xlva>
- *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.
<https://amzn.to/32K9K6d>

21.7.3 APIs

- `sklearn.metrics.roc_curve` API.
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_curve.html
- `imblearn.metrics.geometric_mean_score` API.
https://imbalanced-learn.org/stable/generated/imblearn.metrics.geometric_mean_score.html
- `sklearn.metrics.precision_recall_curve` API.
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_curve.html

21.7.4 Articles

- Discrimination Threshold, Yellowbrick.
<https://www.scikit-yb.org/en/latest/api/classifier/threshold.html>
- Youden's J statistic, Wikipedia.
https://en.wikipedia.org/wiki/Youden%27s_J_statistic
- Receiver operating characteristic, Wikipedia.
https://en.wikipedia.org/wiki/Receiver_operating_characteristic

21.8 Summary

In this tutorial, you discovered how to tune the optimal threshold when converting probabilities to crisp class labels for imbalanced classification. Specifically, you learned:

- The default threshold for interpreting probabilities to class labels is 0.5, and tuning this hyperparameter is called threshold moving.
- How to calculate the optimal threshold for the ROC Curve and Precision-Recall Curve directly.
- How to manually search threshold values for a chosen model and model evaluation metric.

21.8.1 Next

In the next tutorial, you will discover how to calibrate predicted probabilities for imbalanced classification.

Chapter 22

Probability Calibration

Many machine learning models are capable of predicting a probability or probability-like scores for class membership. Probabilities provide a required level of granularity for evaluating and comparing models, especially on imbalanced classification problems where tools like ROC Curves are used to interpret predictions and the ROC AUC metric is used to compare model performance, both of which use probabilities.

Unfortunately, the probabilities or probability-like scores predicted by many models are not calibrated. This means that they may be over-confident in some cases and under-confident in other cases. Worse still, the severely skewed class distribution present in imbalanced classification tasks may result in even more bias in the predicted probabilities as they over-favor predicting the majority class.

As such, it is often a good idea to calibrate the predicted probabilities for nonlinear machine learning models prior to evaluating their performance. Further, it is good practice to calibrate probabilities in general when working with imbalanced datasets, even of models like logistic regression that predict well-calibrated probabilities when the class labels are balanced. In this tutorial, you will discover how to calibrate predicted probabilities for imbalanced classification. After completing this tutorial, you will know:

- Calibrated probabilities are required to get the most out of models for imbalanced classification problems.
- How to calibrate predicted probabilities for nonlinear models like SVMs, decision trees, and KNN.
- How to grid search different probability calibration methods on a dataset with a skewed class distribution.

Let's get started.

22.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Problem of Uncalibrated Probabilities
2. How to Calibrate Probabilities

3. SVM With Calibrated Probabilities
4. Decision Tree With Calibrated Probabilities
5. Grid Search Probability Calibration with KNN

22.2 Problem of Uncalibrated Probabilities

Many machine learning algorithms can predict a probability or a probability-like score that indicates class membership. For example, logistic regression can predict the probability of class membership directly and support vector machines can predict a score that is not a probability but could be interpreted as a probability.

The probability can be used as a measure of uncertainty on those problems where a probabilistic prediction is required. This is particularly the case in imbalanced classification, where crisp class labels are often insufficient both in terms of evaluating and selecting a model. The predicted probability provides the basis for more granular model evaluation and selection, such as through the use of ROC and Precision-Recall diagnostic plots, metrics like ROC AUC, and techniques like threshold moving.

As such, using machine learning models that predict probabilities is generally preferred when working on imbalanced classification tasks. The problem is that few machine learning models have calibrated probabilities.

... to be usefully interpreted as probabilities, the scores should be calibrated.

— Page 57, *Learning from Imbalanced Data Sets*, 2018.

Calibrated probabilities means that the probability reflects the likelihood of true events. This might be confusing if you consider that in classification, we have class labels that are correct or not instead of probabilities. To clarify, recall that in binary classification, we are predicting a negative or positive case as class 0 or 1. If 100 examples are predicted with a probability of 0.8, then 80 percent of the examples will have class 1 and 20 percent will have class 0, if the probabilities are calibrated. Here, calibration is the concordance of predicted probabilities with the occurrence of positive cases. Uncalibrated probabilities suggest that there is a bias in the probability scores, meaning the probabilities are overconfident or under-confident in some cases.

- **Calibrated Probabilities.** Probabilities match the true likelihood of events.
- **Uncalibrated Probabilities.** Probabilities are over-confident and/or under-confident.

This is common for machine learning models that are not trained using a probabilistic framework and for training data that has a skewed distribution, like imbalanced classification tasks. There are two main causes for uncalibrated probabilities; they are:

- Algorithms not trained using a probabilistic framework.
- Biases in the training data.

Few machine learning algorithms produce calibrated probabilities. This is because for a model to predict calibrated probabilities, it must explicitly be trained under a probabilistic framework, such as maximum likelihood estimation. Some examples of algorithms that provide calibrated probabilities include:

- Logistic Regression.
- Linear Discriminant Analysis.
- Naive Bayes.
- Artificial Neural Networks.

Many algorithms either predict a probability-like score or a class label and must be coerced in order to produce a probability-like score. As such, these algorithms often require their *probabilities* to be calibrated prior to use. Examples include:

- Support Vector Machines.
- Decision Trees.
- Ensembles of Decision Trees (bagging, random forest, gradient boosting).
- k -Nearest Neighbors.

A bias in the training dataset, such as a skew in the class distribution, means that the model will naturally predict a higher probability for the majority class than the minority class on average. The problem is, models may overcompensate and give too much focus to the majority class. This even applies to models that typically produce calibrated probabilities like logistic regression.

... class probability estimates attained via supervised learning in imbalanced scenarios systematically underestimate the probabilities for minority class instances, despite ostensibly good overall calibration.

— *Class Probability Estimates are Unreliable for Imbalanced Data (and How to Fix Them)*, 2012.

22.3 How to Calibrate Probabilities

Probabilities are calibrated by rescaling their values so they better match the distribution observed in the training data.

... we desire that the estimated class probabilities are reflective of the true underlying probability of the sample. That is, the predicted class probability (or probability-like value) needs to be well-calibrated. To be well-calibrated, the probabilities must effectively reflect the true likelihood of the event of interest.

— Page 249, *Applied Predictive Modeling*, 2013.

Probability predictions are made on training data and the distribution of probabilities is compared to the expected probabilities and adjusted to provide a better match. This often involves splitting a training dataset and using one portion to train the model and another portion as a validation set to scale the probabilities. There are two main techniques for scaling predicted probabilities; they are Platt scaling and isotonic regression.

- **Platt Scaling.** Logistic regression model to transform probabilities.
- **Isotonic Regression.** Weighted least-squares regression model to transform probabilities.

Platt scaling is a simpler method and was developed to scale the output from a support vector machine to probability values. It involves learning a logistic regression model to perform the transform of scores to calibrated probabilities. Isotonic regression is a more complex weighted least squares regression model. It requires more training data, although it is also more powerful and more general. Here, isotonic simply refers to monotonically increasing mapping of the original probabilities to the rescaled values.

Platt Scaling is most effective when the distortion in the predicted probabilities is sigmoid-shaped. Isotonic Regression is a more powerful calibration method that can correct any monotonic distortion.

— *Predicting Good Probabilities With Supervised Learning*, 2005.

The scikit-learn library provides access to both Platt scaling and isotonic regression methods for calibrating probabilities via the `CalibratedClassifierCV` class. This is a wrapper for a model (like an SVM). The preferred scaling technique is defined via the `method` argument, which can be ‘`sigmoid`’ (Platt scaling) or ‘`isotonic`’ (isotonic regression).

Cross-validation is used to scale the predicted probabilities from the model, set via the `cv` argument. This means that the model is fit on the training set and calibrated on the test set, and this process is repeated k -times for the k -folds where predicted probabilities are averaged across the runs. Setting the `cv` argument depends on the amount of data available, although values such as 3 or 5 can be used. Importantly, the split is stratified, which is important when using probability calibration on imbalanced datasets that often have very few examples of the positive class.

```
...
# example of wrapping a model with probability calibration
model = ...
calibrated = CalibratedClassifierCV(model, method='sigmoid', cv=3)
```

Listing 22.1: Example of defining probability calibration.

Now that we know how to calibrate probabilities, let’s look at some examples of calibrating probability for models on an imbalanced classification dataset.

22.4 SVM With Calibrated Probabilities

In this section, we will review how to calibrate the probabilities for an SVM model on an imbalanced classification dataset. First, let’s define a dataset using the `make_classification()` function. We will generate 10,000 examples, 99 percent of which will belong to the negative case (class 0) and 1 percent will belong to the positive case (class 1).

```
...
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
```

Listing 22.2: Example of defining a binary imbalanced classification dataset.

Next, we can define an SVM with default hyperparameters. This means that the model is not tuned to the dataset, but will provide a consistent basis of comparison.

```
...
# define model
model = SVC(gamma='scale')
```

Listing 22.3: Example of defining an SVM model.

We can then evaluate this model on the dataset using repeated stratified k -fold cross-validation with three repeats of 10-folds. We will evaluate the model using ROC AUC and calculate the mean score across all repeats and folds. The ROC AUC will make use of the uncalibrated probability-like scores provided by the SVM.

```
...
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))
```

Listing 22.4: Example of evaluating the performance of a model.

Tying this together, the complete example is listed below.

```
# evaluate svm with uncalibrated probabilities for imbalanced classification
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.svm import SVC
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
# define model
model = SVC(gamma='scale')
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))
```

Listing 22.5: Example of evaluating an SVM model for imbalanced classification with uncalibrated probabilities.

Running the example evaluates the SVM with uncalibrated probabilities on the imbalanced classification dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the SVM achieved a ROC AUC of about 0.804.

```
Mean ROC AUC: 0.804
```

Listing 22.6: Example output from evaluating an SVM model for imbalanced classification with uncalibrated probabilities.

Next, we can try using the `CalibratedClassifierCV` class to wrap the SVM model and predict calibrated probabilities. We are using stratified 10-fold cross-validation to evaluate the model; that means 9,000 examples are used for train and 1,000 for test on each fold. With `CalibratedClassifierCV` and 3-folds, the 9,000 examples of one fold will be split into 6,000 for training the model and 3,000 for calibrating the probabilities. This does not leave many examples of the minority class, e.g. 90/10 in 10-fold cross-validation, then 60/30 for calibration.

When using calibration, it is important to work through these numbers based on your chosen model evaluation scheme and either adjust the number of folds to ensure the datasets are sufficiently large or even switch to a simpler train/test split instead of cross-validation if needed. Experimentation might be required. We will define the SVM model as before, then define the `CalibratedClassifierCV` with isotonic regression, then evaluate the calibrated model via repeated stratified k -fold cross-validation.

```
...
# define model
model = SVC(gamma='scale')
# wrap the model
calibrated = CalibratedClassifierCV(model, method='isotonic', cv=3)
```

Listing 22.7: Example of defining SVM with calibrated probabilities.

Because SVM probabilities are not calibrated by default, we would expect that calibrating them would result in an improvement to the ROC AUC that explicitly evaluates a model based on their probabilities. Tying this together, the full example below of evaluating SVM with calibrated probabilities is listed below.

```
# evaluate svm with calibrated probabilities for imbalanced classification
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.calibration import CalibratedClassifierCV
from sklearn.svm import SVC
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
# define model
model = SVC(gamma='scale')
# wrap the model
calibrated = CalibratedClassifierCV(model, method='isotonic', cv=3)
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(calibrated, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
```

```
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))
```

Listing 22.8: Example of evaluating an SVM model for imbalanced classification with calibrated probabilities.

Running the example evaluates the SVM with calibrated probabilities on the imbalanced classification dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the SVM achieved a lift in ROC AUC from about 0.804 to about 0.875.

```
Mean ROC AUC: 0.875
```

Listing 22.9: Example output from evaluating an SVM model for imbalanced classification with calibrated probabilities.

Probability calibration can be evaluated in conjunction with other modifications to the algorithm or dataset to address the skewed class distribution. For example, SVM provides the `class_weight` argument that can be set to ‘`balanced`’ to adjust the margin to favor the minority class. We can include this change to SVM and calibrate the probabilities, and we might expect to see a further lift in model skill; for example:

```
...
# define model
model = SVC(gamma='scale', class_weight='balanced')
```

Listing 22.10: Example of defining SVM with balanced class weighting.

Tying this together, the complete example of a class weighted SVM with calibrated probabilities is listed below.

```
# evaluate weighted svm with calibrated probabilities for imbalanced classification
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.calibration import CalibratedClassifierCV
from sklearn.svm import SVC
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
# define model
model = SVC(gamma='scale', class_weight='balanced')
# wrap the model
calibrated = CalibratedClassifierCV(model, method='isotonic', cv=3)
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(calibrated, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))
```

Listing 22.11: Example of evaluating an balanced class-weight SVM model for imbalanced classification with calibrated probabilities.

Running the example evaluates the class-weighted SVM with calibrated probabilities on the imbalanced classification dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the SVM achieved a further lift in ROC AUC from about 0.875 to about 0.966.

```
Mean ROC AUC: 0.966
```

Listing 22.12: Example output from evaluating an balanced class-weight SVM model for imbalanced classification with calibrated probabilities.

22.5 Decision Tree With Calibrated Probabilities

Decision trees are another highly effective machine learning algorithm that does not naturally produce probabilities. Instead, class labels are predicted directly and a probability-like score can be estimated based on the distribution of examples in the training dataset that fall into the leaf of the tree that is predicted for the new example. As such, the probability scores from a decision tree should be calibrated prior to being evaluated and used to select a model.

We can define a decision tree using the `DecisionTreeClassifier` scikit-learn class. The model can be evaluated with uncalibrated probabilities on our synthetic imbalanced classification dataset. The complete example is listed below.

```
# evaluate decision tree with uncalibrated probabilities for imbalanced classification
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.tree import DecisionTreeClassifier
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
# define model
model = DecisionTreeClassifier()
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))
```

Listing 22.13: Example of evaluating an decision tree model for imbalanced classification with uncalibrated probabilities.

Running the example evaluates the decision tree with uncalibrated probabilities on the imbalanced classification dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the decision tree achieved a ROC AUC of about 0.842.

```
Mean ROC AUC: 0.842
```

Listing 22.14: Example output from evaluating an decision tree model for imbalanced classification with uncalibrated probabilities.

We can then evaluate the same model using the calibration wrapper. In this case, we will use the Platt Scaling method configured by setting the `method` argument to ‘`sigmoid`’.

```
...
# define model
model = DecisionTreeClassifier()
# wrap the model
calibrated = CalibratedClassifierCV(model, method='sigmoid', cv=3)
```

Listing 22.15: Example of defining probability calibration for the decision tree.

The complete example of evaluating the decision tree with calibrated probabilities for imbalanced classification is listed below.

```
# decision tree with calibrated probabilities for imbalanced classification
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.calibration import CalibratedClassifierCV
from sklearn.tree import DecisionTreeClassifier
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
# define model
model = DecisionTreeClassifier()
# wrap the model
calibrated = CalibratedClassifierCV(model, method='sigmoid', cv=3)
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(calibrated, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))
```

Listing 22.16: Example of evaluating an decision tree model for imbalanced classification with calibrated probabilities.

Running the example evaluates the decision tree with calibrated probabilities on the imbalanced classification dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the decision tree achieved a lift in ROC AUC from about 0.842 to about 0.859.

```
Mean ROC AUC: 0.859
```

Listing 22.17: Example output from evaluating an decision tree model for imbalanced classification with calibrated probabilities.

22.6 Grid Search Probability Calibration With KNN

Probability calibration can be sensitive to both the method and the way in which the method is employed. As such, it is a good idea to test a suite of different probability calibration methods on your model in order to discover what works best for your dataset. One approach is to treat the calibration method and cross-validation folds as hyperparameters and tune them. In this section, we will look at using a grid search to tune these hyperparameters.

The *k*-nearest neighbor, or KNN, algorithm is another nonlinear machine learning algorithm that predicts a class label directly and must be modified to produce a probability-like score. This often involves using the distribution of class labels in the neighborhood. We can evaluate a KNN with uncalibrated probabilities on our synthetic imbalanced classification dataset using the `KNeighborsClassifier` class with a default neighborhood size of 5. The complete example is listed below.

```
# evaluate knn with uncalibrated probabilities for imbalanced classification
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.neighbors import KNeighborsClassifier
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
# define model
model = KNeighborsClassifier()
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))
```

Listing 22.18: Example of evaluating a KNN model for imbalanced classification with uncalibrated probabilities.

Running the example evaluates the KNN with uncalibrated probabilities on the imbalanced classification dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the KNN achieved a ROC AUC of about 0.864.

```
Mean ROC AUC: 0.864
```

Listing 22.19: Example output from evaluating a KNN model for imbalanced classification with uncalibrated probabilities.

Knowing that the probabilities are dependent on the neighborhood size and are uncalibrated, we would expect that some calibration would improve the performance of the model using ROC AUC. Rather than spot-checking one configuration of the `CalibratedClassifierCV` class, we will instead use the `GridSearchCV` to grid search different configurations. First, the model and calibration wrapper are defined as before.

```
...
# define model
model = KNeighborsClassifier()
# wrap the model
calibrated = CalibratedClassifierCV(model)
```

Listing 22.20: Example of defining probability calibration for the KNN.

We will test both ‘sigmoid’ and ‘isotonic’ values for the `method` argument, and different `cv` values in [2,3,4]. Recall that `cv` controls the split of the training dataset that is used to estimate the calibrated probabilities. We can define the grid of parameters as a dict with the names of the arguments to the `CalibratedClassifierCV` we want to tune and provide lists of values to try. This will test 3×2 (6) different combinations.

```
...
# define grid
param_grid = dict(cv=[2,3,4], method=['sigmoid', 'isotonic'])
```

Listing 22.21: Example of defining a grid of configurations to evaluate.

We can then define the `GridSearchCV` with the model and grid of parameters and use the same repeated stratified k -fold cross-validation we used before to evaluate each parameter combination.

```
...
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# define grid search
grid = GridSearchCV(estimator=calibrated, param_grid=param_grid, n_jobs=-1, cv=cv,
    scoring='roc_auc')
# execute the grid search
grid_result = grid.fit(X, y)
```

Listing 22.22: Example of defining and executing the grid search.

Once evaluated, we will then summarize the configuration found with the highest ROC AUC, then list the results for all combinations.

```
# report the best configuration
print('Best: %f using %s' % (grid_result.best_score_, grid_result.best_params_))
# report all configurations
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print('%f (%f) with: %r' % (mean, stdev, param))
```

Listing 22.23: Example of reporting the results from the grid search.

Tying this together, the complete example of grid searching probability calibration for imbalanced classification with a KNN model is listed below.

```
# grid search probability calibration with knn for imbalanced classification
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.neighbors import KNeighborsClassifier
from sklearn.calibration import CalibratedClassifierCV
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
# define model
model = KNeighborsClassifier()
# wrap the model
calibrated = CalibratedClassifierCV(model)
# define grid
param_grid = dict(cv=[2,3,4], method=['sigmoid','isotonic'])
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# define grid search
grid = GridSearchCV(estimator=calibrated, param_grid=param_grid, n_jobs=-1, cv=cv,
                     scoring='roc_auc')
# execute the grid search
grid_result = grid.fit(X, y)
# report the best configuration
print('Best: %f using %s' % (grid_result.best_score_, grid_result.best_params_))
# report all configurations
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print('%f (%f) with: %r' % (mean, stdev, param))
```

Listing 22.24: Example of grid searching probability calibration methods for the KNN model.

Running the example evaluates the KNN with a suite of different types of calibrated probabilities on the imbalanced classification dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the best result was achieved with a `cv` of 2 and an ‘`isotonic`’ value for `method` achieving a mean ROC AUC of about 0.895, a lift from 0.864 achieved with no calibration.

```
Best: 0.895120 using {'cv': 2, 'method': 'isotonic'}
0.895084 (0.062358) with: {'cv': 2, 'method': 'sigmoid'}
0.895120 (0.062488) with: {'cv': 2, 'method': 'isotonic'}
0.885221 (0.061373) with: {'cv': 3, 'method': 'sigmoid'}
0.881924 (0.064351) with: {'cv': 3, 'method': 'isotonic'}
0.881865 (0.065708) with: {'cv': 4, 'method': 'sigmoid'}
0.875320 (0.067663) with: {'cv': 4, 'method': 'isotonic'}
```

Listing 22.25: Example output from grid searching probability calibration methods for the KNN model.

This provides a template that you can use to evaluate different probability calibration configurations on your own models.

22.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

22.7.1 Papers

- *Predicting Good Probabilities With Supervised Learning*, 2005.
<https://dl.acm.org/citation.cfm?id=1102430>
- *Class Probability Estimates are Unreliable for Imbalanced Data (and How to Fix Them)*, 2012.
<https://ieeexplore.ieee.org/abstract/document/6413859>

22.7.2 Books

- *Learning from Imbalanced Data Sets*, 2018.
<https://amzn.to/307Xlva>
- *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.
<https://amzn.to/32K9K6d>
- *Applied Predictive Modeling*, 2013.
<https://amzn.to/2kXE35G>

22.7.3 APIs

- `sklearn.calibration.CalibratedClassifierCV` API.
<https://scikit-learn.org/stable/modules/generated/sklearn.calibration.CalibratedClassifierCV.html>
- `sklearn.svm.SVC` API.
<https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html>
- `sklearn.tree.DecisionTreeClassifier` API.
<https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html>
- `sklearn.neighbors.KNeighborsClassifier` API.
<https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html>
- `sklearn.model_selection.GridSearchCV` API.
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

22.7.4 Articles

- Calibration (statistics), Wikipedia.
[https://en.wikipedia.org/wiki/Calibration_\(statistics\)](https://en.wikipedia.org/wiki/Calibration_(statistics))
- Probabilistic classification, Wikipedia.
https://en.wikipedia.org/wiki/Probabilistic_classification
- Platt scaling, Wikipedia.
https://en.wikipedia.org/wiki/Platt_scaling
- Isotonic regression, Wikipedia.
https://en.wikipedia.org/wiki/Isotonic_regression

22.8 Summary

In this tutorial, you discovered how to calibrate predicted probabilities for imbalanced classification. Specifically, you learned:

- Calibrated probabilities are required to get the most out of models for imbalanced classification problems.
- How to calibrate predicted probabilities for nonlinear models like SVMs, decision trees, and KNN.
- How to grid search different probability calibration methods on datasets with a skewed class distribution.

22.8.1 Next

In the next tutorial, you will discover how to use specialized ensemble algorithms for imbalanced classification.

Chapter 23

Ensemble Algorithms

Bagging is an ensemble algorithm that fits multiple models on different subsets of a training dataset, then combines the predictions from all models. Random forests are an extension of bagging that also randomly selects subsets of features used in each data sample. Both bagging and random forests have proven effective on a wide range of different predictive modeling problems.

Although effective, they are not suited to classification problems with a skewed class distribution. Nevertheless, many modifications to the algorithms have been proposed that adapt their behavior and make them better suited to a severe class imbalance. In this tutorial, you will discover how to use bagging and random forest for imbalanced classification. After completing this tutorial, you will know:

- How to use Bagging with random undersampling for imbalanced classification.
- How to use Random Forest with class weighting and random undersampling for imbalanced classification.
- How to use the Easy Ensemble that combines bagging and boosting for imbalanced classification.

Let's get started.

Note: This chapter makes use of the imbalanced-learn library. See Appendix [B](#) for installation instructions, if needed.

23.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Bagging for Imbalanced Classification
2. Random Forest for Imbalanced Classification
3. Easy Ensemble for Imbalanced Classification

23.2 Bagging for Imbalanced Classification

Bootstrap Aggregation, or Bagging for short, is an ensemble machine learning algorithm. It involves first selecting random samples from the training dataset with replacement, meaning that a given sample may contain zero, one, or more than one copy of examples in the training dataset. This is called a bootstrap sample. One weak learner model is then fit on each data sample. Typically, decision tree models that do not use pruning (e.g. may overfit their training set slightly) are used as weak learners. Finally, the predictions from all of the fit weak learners are combined to make a single prediction (e.g. aggregated).

Each model in the ensemble is then used to generate a prediction for a new sample and these m predictions are averaged to give the bagged model's prediction.

— Page 192, *Applied Predictive Modeling*, 2013.

The process of creating new bootstrap samples and fitting and adding trees to the sample can continue until no further improvement is seen in the ensemble's performance on a validation dataset. This simple procedure often results in better performance than a single well-configured decision tree algorithm. Bagging as-is will create bootstrap samples that will not consider the skewed class distribution for imbalanced classification datasets. As such, although the technique performs well in general, it may not perform well if a severe class imbalance is present.

23.2.1 Standard Bagging

Before we dive into exploring extensions to bagging, let's evaluate a standard bagged decision tree ensemble and use it as a point of comparison. We can use the `BaggingClassifier` scikit-learn class to create a bagged decision tree model. First, let's define a synthetic imbalanced binary classification problem with 10,000 examples, 99 percent of which are in the majority class and 1 percent are in the minority class.

```
...
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
```

Listing 23.1: Example of defining a binary imbalanced classification dataset.

We can then define the standard bagged decision tree ensemble model ready for evaluation.

```
...
# define model
model = BaggingClassifier()
```

Listing 23.2: Example of defining the bagging model.

We can then evaluate this model using repeated stratified k -fold cross-validation, with three repeats and 10 folds. We will use the mean ROC AUC score across all folds and repeats to evaluate the performance of the model.

```
...
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
```

```
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
```

Listing 23.3: Example of defining the model evaluation procedure.

Tying this together, the complete example of evaluating a standard bagged ensemble on the imbalanced classification dataset is listed below.

```
# bagged decision trees on an imbalanced classification problem
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.ensemble import BaggingClassifier
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
# define model
model = BaggingClassifier()
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))
```

Listing 23.4: Example of evaluating standard bagging on the imbalanced classification dataset.

Running the example evaluates the model and reports the mean ROC AUC score.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the model achieves a score of about 0.87.

```
Mean ROC AUC: 0.871
```

Listing 23.5: Example output from evaluating standard bagging on the imbalanced classification dataset.

23.2.2 Bagging With Random Undersampling

There are many ways to adapt bagging for use with imbalanced classification. Perhaps the most straightforward approach is to apply data sampling on the bootstrap sample prior to fitting the weak learner model. This might involve oversampling the minority class or undersampling the majority class.

An easy way to overcome class imbalance problem when facing the resampling stage in bagging is to take the classes of the instances into account when they are randomly drawn from the original dataset.

Oversampling the minority class in the bootstrap is referred to as OverBagging; likewise, undersampling the majority class in the bootstrap is referred to as UnderBagging, and combining both approaches is referred to as OverUnderBagging. The imbalanced-learn library provides an implementation of UnderBagging. Specifically, it provides a version of bagging that uses a random undersampling strategy on the majority class within a bootstrap sample in order to balance the two classes. This is provided in the `BalancedBaggingClassifier` class.

```
...
# define model
model = BalancedBaggingClassifier()
```

Listing 23.6: Example of defining the balanced bagging model.

Next, we can evaluate a modified version of the bagged decision tree ensemble that performs random undersampling of the majority class prior to fitting each decision tree. We would expect that the use of random undersampling would improve the performance of the ensemble. The default number of trees (`n_estimators`) for this model and the previous is 10. In practice, it is a good idea to test larger values for this hyperparameter, such as 100 or 1,000. The complete example is listed below.

```
# bagged decision trees with random undersampling for imbalanced classification
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from imblearn.ensemble import BalancedBaggingClassifier
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
# define model
model = BalancedBaggingClassifier()
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))
```

Listing 23.7: Example of evaluating balanced bagging on the imbalanced classification dataset.

Running the example evaluates the model and reports the mean ROC AUC score.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see a lift on mean ROC AUC from about 0.87 without any data sampling, to about 0.96 with random undersampling of the majority class. This is not a true apples-to-apples comparison as we are using the same algorithm implementation from two different libraries, but it makes the general point that balancing the bootstrap prior to fitting a weak learner offers some benefit when the class distribution is skewed.

```
Mean ROC AUC: 0.962
```

Listing 23.8: Example output from evaluating balanced bagging on the imbalanced classification dataset.

Although the `BalancedBaggingClassifier` class uses a decision tree, you can test different models, such as k -nearest neighbors and more. You can set the `base_estimator` argument when defining the class to use a different weak learner classifier model.

23.3 Random Forest for Imbalanced Classification

Random forest is another ensemble of decision tree models and may be considered an improvement upon bagging. Like bagging, random forest involves selecting bootstrap samples from the training dataset and fitting a decision tree on each. The main difference is that all features (variables or columns) are not used; instead, a small, randomly selected subset of features (columns) is chosen for each bootstrap sample. This has the effect of de-correlating the decision trees (making them more independent), and in turn, improving the ensemble prediction.

Each model in the ensemble is then used to generate a prediction for a new sample and these m predictions are averaged to give the forest's prediction. Since the algorithm randomly selects predictors at each split, tree correlation will necessarily be lessened.

— Page 199, *Applied Predictive Modeling*, 2013.

Again, random forest is very effective on a wide range of problems, but like bagging, performance of the standard algorithm is not great on imbalanced classification problems.

In learning extremely imbalanced data, there is a significant probability that a bootstrap sample contains few or even none of the minority class, resulting in a tree with poor performance for predicting the minority class.

— *Using Random Forest to Learn Imbalanced Data*, 2004.

23.3.1 Standard Random Forest

Before we dive into extensions of the random forest ensemble algorithm to make it better suited for imbalanced classification, let's fit and evaluate a random forest algorithm on our synthetic dataset. We can use the `RandomForestClassifier` class from scikit-learn and use a small number of trees, in this case, 10.

```
...
# define model
model = RandomForestClassifier(n_estimators=10)
```

Listing 23.9: Example of defining the standard random forest model.

The complete example of fitting a standard random forest ensemble on the imbalanced dataset is listed below.

```
# random forest for imbalanced classification
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
```

```

from sklearn.ensemble import RandomForestClassifier
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
# define model
model = RandomForestClassifier(n_estimators=10)
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))

```

Listing 23.10: Example of evaluating standard random forest on the imbalanced classification dataset.

Running the example evaluates the model and reports the mean ROC AUC score.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the model achieved a mean ROC AUC of about 0.86.

```
Mean ROC AUC: 0.869
```

Listing 23.11: Example output from evaluating standard random forest on the imbalanced classification dataset.

23.3.2 Random Forest With Class Weighting

A simple technique for modifying a decision tree for imbalanced classification is to change the weight that each class has when calculating the *impurity* score of a chosen split point. Impurity measures how mixed the groups of samples are for a given split in the training dataset and is typically measured with Gini or entropy. The calculation can be biased so that a mixture in favor of the minority class is favored, allowing some false positives for the majority class. This modification of random forest is referred to as Weighted Random Forest.

Another approach to make random forest more suitable for learning from extremely imbalanced data follows the idea of cost sensitive learning. Since the RF classifier tends to be biased towards the majority class, we shall place a heavier penalty on misclassifying the minority class.

— *Using Random Forest to Learn Imbalanced Data*, 2004.

This can be achieved by setting the `class_weight` argument on the `RandomForestClassifier` class. This argument takes a dictionary with a mapping of each class value (e.g. 0 and 1) to the weighting. The argument value of ‘`balanced`’ can be provided to automatically use the inverse weighting from the training dataset, giving focus to the minority class.

```

...
# define model
model = RandomForestClassifier(n_estimators=10, class_weight='balanced')

```

Listing 23.12: Example of defining a balanced random forest model.

We can test this modification of random forest on our test problem. Although not specific to random forest, we would expect some modest improvement. The complete example is listed below.

```
# class balanced random forest for imbalanced classification
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.ensemble import RandomForestClassifier
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
# define model
model = RandomForestClassifier(n_estimators=10, class_weight='balanced')
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))
```

Listing 23.13: Example of evaluating balanced random forest on the imbalanced classification dataset.

Running the example evaluates the model and reports the mean ROC AUC score.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the model achieved a modest lift in mean ROC AUC from 0.86 to about 0.87.

Mean ROC AUC: 0.871

Listing 23.14: Example output from evaluating balanced random forest on the imbalanced classification dataset.

23.3.3 Random Forest With Bootstrap Class Weighting

Given that each decision tree is constructed from a bootstrap sample (e.g. random selection with replacement), the class distribution in the data sample will be different for each tree. As such, it might be interesting to change the class weighting based on the class distribution in each bootstrap sample, instead of the entire training dataset. This can be achieved by setting the `class_weight` argument to the value ‘`balanced_subsample`’. We can test this modification and compare the results to the ‘`balanced`’ case above; the complete example is listed below.

```
# bootstrap class balanced random forest for imbalanced classification
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.ensemble import RandomForestClassifier
# generate dataset
```

```

X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
# define model
model = RandomForestClassifier(n_estimators=10, class_weight='balanced_subsample')
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))

```

Listing 23.15: Example of evaluating bootstrap balanced random forest on the imbalanced classification dataset.

Running the example evaluates the model and reports the mean ROC AUC score.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the model achieved a modest lift in mean ROC AUC from 0.87 to about 0.88.

```
Mean ROC AUC: 0.884
```

Listing 23.16: Example output from evaluating bootstrap balanced random forest on the imbalanced classification dataset.

23.3.4 Random Forest With Random Undersampling

Another useful modification to random forest is to perform data sampling on the bootstrap sample in order to explicitly change the class distribution. The `BalancedRandomForestClassifier` class from the imbalanced-learn library implements this and performs random undersampling of the majority class in each bootstrap sample. This is generally referred to as Balanced Random Forest.

```

...
# define model
model = BalancedRandomForestClassifier(n_estimators=10)

```

Listing 23.17: Example of defining a random forest model with undersampling.

We would expect this to have a more dramatic effect on model performance, given the broader success of data sampling techniques. We can test this modification of random forest on our synthetic dataset and compare the results. The complete example is listed below.

```

# random forest with random undersampling for imbalanced classification
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from imblearn.ensemble import BalancedRandomForestClassifier
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)

```

```
# define model
model = BalancedRandomForestClassifier(n_estimators=10)
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))
```

Listing 23.18: Example of evaluating random forest with undersampling on the imbalanced classification dataset.

Running the example evaluates the model and reports the mean ROC AUC score.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the model achieved a lift in mean ROC AUC from 0.89 to about 0.97.

```
Mean ROC AUC: 0.970
```

Listing 23.19: Example output from evaluating random forest with undersampling on the imbalanced classification dataset.

23.4 Easy Ensemble for Imbalanced Classification

When considering bagged ensembles for imbalanced classification, a natural thought might be to use random sampling of the majority class to create multiple datasets with a balanced class distribution. Specifically, a dataset can be created from all of the examples in the minority class and a randomly selected sample from the majority class. Then a model or weak learner can be fit on this dataset. The process can be repeated multiple times and the average prediction across the ensemble of models can be used to make predictions.

This is exactly the approach proposed by Xu-Ying Liu, et al. in their 2008 paper titled *Exploratory Undersampling for Class-Imbalance Learning*. The selective construction of the subsamples is seen as a type of undersampling of the majority class. The generation of multiple subsamples allows the ensemble to overcome the downside of undersampling in which valuable information is discarded from the training process.

... under-sampling is an efficient strategy to deal with class-imbalance. However, the drawback of under-sampling is that it throws away many potentially useful data.

— *Exploratory Undersampling for Class-Imbalance Learning*, 2008.

The authors propose variations on the approach, such as the Easy Ensemble and the Balance Cascade. Let's take a closer look at the Easy Ensemble.

23.4.1 Easy Ensemble

The Easy Ensemble involves creating balanced samples of the training dataset by selecting all examples from the minority class and a subset from the majority class. Rather than using pruned decision trees, boosted decision trees are used on each subset, specifically the AdaBoost algorithm. AdaBoost works by first fitting a decision tree on the dataset, then determining the errors made by the tree and weighing the examples in the dataset by those errors so that more attention is paid to the misclassified examples and less to the correctly classified examples. A subsequent tree is then fit on the weighted dataset intended to correct the errors. The process is then repeated for a given number of decision trees.

This means that samples that are difficult to classify receive increasingly larger weights until the algorithm identifies a model that correctly classifies these samples. Therefore, each iteration of the algorithm is required to learn a different aspect of the data, focusing on regions that contain difficult-to-classify samples.

— Page 389, *Applied Predictive Modeling*, 2013.

The `EasyEnsembleClassifier` class from the imbalanced-learn library provides an implementation of the easy ensemble technique.

```
...
# define model
model = EasyEnsembleClassifier(n_estimators=10)
```

Listing 23.20: Example of defining the easy ensemble classifier.

We can evaluate the technique on our synthetic imbalanced classification problem. Given that the model performs a type of random undersampling, we would expect the technique to perform well in general. The complete example is listed below.

```
# easy ensemble for imbalanced classification
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from imblearn.ensemble import EasyEnsembleClassifier
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=4)
# define model
model = EasyEnsembleClassifier(n_estimators=10)
# define evaluation procedure
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# evaluate model
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
# summarize performance
print('Mean ROC AUC: %.3f' % mean(scores))
```

Listing 23.21: Example of evaluating the easy ensemble on the imbalanced classification dataset.

Running the example evaluates the model and reports the mean ROC AUC score.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the ensemble performs well on the dataset, achieving a mean ROC AUC of about 0.96, close to that achieved on this dataset with random forest with random undersampling (0.97).

```
Mean ROC AUC: 0.968
```

Listing 23.22: Example output from evaluating the easy ensemble on the imbalanced classification dataset.

Although an AdaBoost classifier is used on each subsample, alternate classifier models can be used via setting the `base_estimator` argument to the model.

23.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

23.5.1 Papers

- *Using Random Forest to Learn Imbalanced Data*, 2004.
<https://statistics.berkeley.edu/tech-reports/666>
- *Exploratory Undersampling for Class-Imbalance Learning*, 2008.
<https://ieeexplore.ieee.org/document/4717268>

23.5.2 Books

- *Learning from Imbalanced Data Sets*, 2018.
<https://amzn.to/307Xlva>
- *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.
<https://amzn.to/32K9K6d>
- *Applied Predictive Modeling*, 2013.
<https://amzn.to/2W8wnPS>

23.5.3 APIs

- `imblearn.ensemble.BalancedBaggingClassifier` API.
<https://imbalanced-learn.org/stable/generated/imblearn.ensemble.BalancedBaggingClassifier.html>
- `sklearn.ensemble.BaggingClassifier` API.
<https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html>
- `sklearn.ensemble.RandomForestClassifier` API.
<https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>

- `imblearn.ensemble.BalancedRandomForestClassifier` API.

<https://imbalanced-learn.org/stable/generated/imblearn.ensemble.BalancedRandomForestClassifier.html>

23.6 Summary

In this tutorial, you discovered how to use bagging and random forest for imbalanced classification. Specifically, you learned:

- How to use Bagging with random undersampling for imbalance classification.
- How to use Random Forest with class weighting and random undersampling for imbalanced classification.
- How to use the Easy Ensemble that combines bagging and boosting for imbalanced classification.

23.6.1 Next

In the next tutorial, you will discover how to use outlier detection algorithms for imbalanced classification.

Chapter 24

One-Class Classification

Outliers or anomalies are rare examples that do not fit in with the rest of the data. Identifying outliers in data is referred to as outlier or anomaly detection and a subfield of machine learning focused on this problem is referred to as one-class classification. These are unsupervised learning algorithms that attempt to model *normal* examples in order to classify new examples as either normal or abnormal (e.g. outliers). One-class classification algorithms can be used for binary classification tasks with a severely skewed class distribution. These techniques can be fit on the input examples from the majority class in the training dataset, then evaluated on a holdout test dataset.

Although not designed for these types of problems, one-class classification algorithms can be effective for imbalanced classification datasets where there are none or very few examples of the minority class, or datasets where there is no coherent structure to separate the classes that could be learned by a supervised algorithm. In this tutorial, you will discover how to use one-class classification algorithms for datasets with severely skewed class distributions. After completing this tutorial, you will know:

- One-class classification is a field of machine learning that provides techniques for outlier and anomaly detection.
- How to adapt one-class classification algorithms for imbalanced classification with a severely skewed class distribution.
- How to fit and evaluate one-class classification algorithms such as SVM, isolation forest, elliptic envelope, and local outlier factor.

Let's get started.

24.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. One-Class Classification for Imbalanced Data
2. One-Class Support Vector Machines
3. Isolation Forest

4. Minimum Covariance Determinant
5. Local Outlier Factor

24.2 One-Class Classification for Imbalanced Data

Outliers are both rare and unusual. Rarity suggests that they have a low frequency relative to non-outlier data (so-called inliers). Unusual suggests that they do not fit neatly into the data distribution. The presence of outliers can cause problems. For example, a single variable may have an outlier far from the mass of examples, which can skew summary statistics such as the mean and variance. Fitting a machine learning model may require the identification and removal of outliers as a data preparation technique.

The process of identifying outliers in a dataset is generally referred to as anomaly detection, where the outliers are *anomalies*, and the rest of the data is *normal*. Outlier detection or anomaly detection is a challenging problem and is comprised of a range of techniques. In machine learning, one approach to tackling the problem of anomaly detection is one-class classification. One-Class Classification, or OCC for short, involves fitting a model on the *normal* data and predicting whether new data is normal or an outlier/anomaly.

A one-class classifier aims at capturing characteristics of training instances, in order to be able to distinguish between them and potential outliers to appear.

— Page 139, *Learning from Imbalanced Data Sets*, 2018.

A one-class classifier is fit on a training dataset that only has examples from the normal class. Once prepared, the model is used to classify new examples as either normal or not-normal, i.e. outliers or anomalies. One-class classification techniques can be used for binary (two-class) imbalanced classification problems where the negative case (class 0) is taken as *normal* and the positive case (class 1) is taken as an outlier or anomaly.

- **Negative Case:** Normal or inlier.
- **Positive Case:** Anomaly or outlier.

Given the nature of the approach, one-class classifications are most suited for those tasks where the positive cases don't have a consistent pattern or structure in the feature space, making it hard for other classification algorithms to learn a class boundary. Instead, treating the positive cases as outliers, it allows one-class classifiers to ignore the task of discrimination and instead focus on deviations from normal or what is expected.

This solution has proven to be especially useful when the minority class lack any structure, being predominantly composed of small disjuncts or noisy instances.

— Page 139, *Learning from Imbalanced Data Sets*, 2018.

It may also be appropriate where the number of positive cases in the training set is so few that they are not worth including in the model, such as a few tens of examples or fewer. Or for problems where no examples of positive cases can be collected prior to training a model.

To be clear, this adaptation of one-class classification algorithms for imbalanced classification is unusual but can be effective on some problems. The downside of this approach is that any examples of outliers (positive cases) we have during training are not used by the one-class classifier and are discarded. This suggests that perhaps an inverse modeling of the problem (e.g. model the positive case as normal) could be tried in parallel. It also suggests that the one-class classifier could provide an input to an ensemble of algorithms, each of which uses the training dataset in different ways.

One must remember that the advantages of one-class classifiers come at a price of discarding all of available information about the minority class. Therefore, this solution should be used carefully and may not fit some specific applications.

— Page 140, *Learning from Imbalanced Data Sets*, 2018.

The scikit-learn library provides a handful of common one-class classification algorithms intended for use in outlier or anomaly detection and change detection, such as One-Class SVM, Isolation Forest, Elliptic Envelope, and Local Outlier Factor. In the following sections, we will take a look at each in turn.

Before we do, we will devise a binary classification dataset to demonstrate the algorithms. We will use the `make_classification()` scikit-learn function to create 10,000 examples with 10 examples in the minority class and 9,990 in the majority class, or a 0.1 percent vs. 99.9 percent, or about 1:1000 class distribution. The example below creates and summarizes this dataset.

```
# Generate and plot a synthetic imbalanced classification dataset
from collections import Counter
from sklearn.datasets import make_classification
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.999], flip_y=0, random_state=4)
# summarize class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
    row_ix = where(y == label)[0]
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Listing 24.1: Example of defining and summarizing the severely imbalanced classification dataset.

Running the example first summarizes the class distribution, confirming the imbalance was created as expected.

```
Counter({0: 9990, 1: 10})
```

Listing 24.2: Example output from defining and summarizing the severely imbalanced classification dataset.

Next, a scatter plot is created and examples are plotted as points colored by their class label, showing a large mass for the majority class (blue) and a few dots for the minority class (orange). This severe class imbalance with so few examples in the positive class and the unstructured nature of the few examples in the positive class might make a good basis for using one-class classification methods.

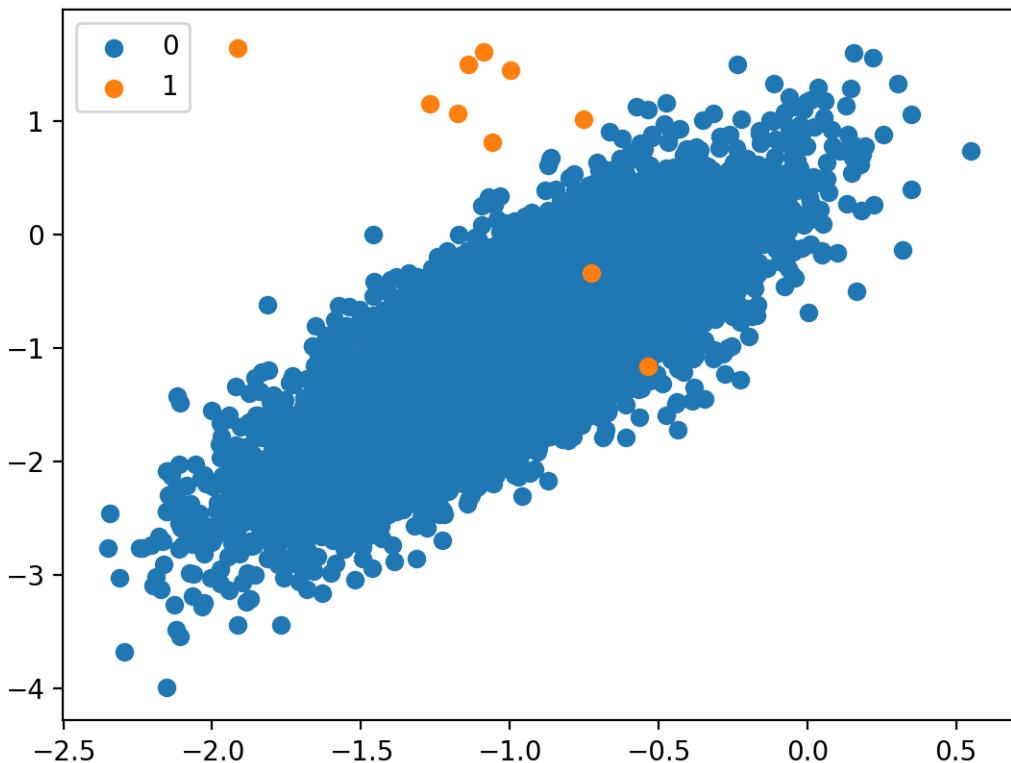


Figure 24.1: Scatter Plot of a Binary Classification Problem With a 1 to 1000 Class Imbalance.

24.3 One-Class Support Vector Machines

The support vector machine, or SVM, algorithm developed initially for binary classification can be used for one-class classification. If used for imbalanced classification, it is a good idea to evaluate the standard SVM and weighted SVM on your dataset before testing the one-class version. When modeling one class, the algorithm captures the density of the majority class and classifies examples on the extremes of the density function as outliers. This modification of SVM is referred to as One-Class SVM.

... an algorithm that computes a binary function that is supposed to capture regions in input space where the probability density lives (its support), that is, a function such that most of the data will live in the region where the function is nonzero.

— *Estimating the Support of a High-Dimensional Distribution*, 2001.

The scikit-learn library provides an implementation of one-class SVM in the `OneClassSVM` class. The main difference from a standard SVM is that it is fit in an unsupervised manner and does not provide the normal hyperparameters for tuning the margin like `C`. Instead, it provides a hyperparameter `nu` that controls the sensitivity of the support vectors and should be tuned to the approximate ratio of outliers in the data, e.g. 0.01%.

```
...
# define outlier detection model
model = OneClassSVM(gamma='scale', nu=0.01)
```

Listing 24.3: Example of defining the one class SVM model.

The model can be fit on all examples in the training dataset or just those examples in the majority class. Perhaps try both on your problem. In this case, we will try fitting on just those examples in the training set that belong to the majority class.

```
# fit on majority class
trainX = trainX[trainy==0]
model.fit(trainX)
```

Listing 24.4: Example of fitting the one class SVM model.

Once fit, the model can be used to identify outliers in new data. When calling the `predict()` function on the model, it will output a +1 for normal examples, so-called inliers, and a -1 for outliers.

- **Inlier Prediction:** +1
- **Outlier Prediction:** -1

```
...
# detect outliers in the test set
yhat = model.predict(testX)
```

Listing 24.5: Example of making a prediction with the one class SVM model.

If we want to evaluate the performance of the model as a binary classifier, we must change the labels in the test dataset from 0 and 1 for the majority and minority classes respectively, to +1 and -1.

```
...
# mark inliers 1, outliers -1
testy[testy == 1] = -1
testy[testy == 0] = 1
```

Listing 24.6: Example of mapping class labels to the expected encoding.

We can then compare the predictions from the model to the expected target values and calculate a score. Given that we have crisp class labels, we might use a score like precision, recall, or a combination of both, such as the F-measure. In this case, we will use F-measure score, which is the harmonic mean of precision and recall. We can calculate the F-measure using the `f1_score()` function and specify the label of the minority class as -1 via the `pos_label` argument.

```

...
# calculate score
score = f1_score(testy, yhat, pos_label=-1)
print('F-measure: %.3f' % score)

```

Listing 24.7: Example of evaluating predictions made by the one class SVM model.

Tying this together, we can evaluate the one-class SVM algorithm on our synthetic dataset. We will split the dataset in two and use half to train the model in an unsupervised manner and the other half to evaluate it. The complete example is listed below.

```

# one-class svm for imbalanced binary classification
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import f1_score
from sklearn.svm import OneClassSVM
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, weights=[0.999], flip_y=0, random_state=4)
# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2,
                                                stratify=y)
# define outlier detection model
model = OneClassSVM(gamma='scale', nu=0.01)
# fit on majority class
trainX = trainX[trainy==0]
model.fit(trainX)
# detect outliers in the test set
yhat = model.predict(testX)
# mark inliers 1, outliers -1
testy[testy == 1] = -1
testy[testy == 0] = 1
# calculate score
score = f1_score(testy, yhat, pos_label=-1)
print('F-measure: %.3f' % score)

```

Listing 24.8: Example of evaluating the one class SVM model on the severely imbalanced classification dataset.

Running the example fits the model on the input examples from the majority class in the training set. The model is then used to classify examples in the test set as inliers and outliers.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, an F-measure score of 0.123 is achieved.

```
F-measure: 0.123
```

Listing 24.9: Example output from evaluating the one class SVM model on the severely imbalanced classification dataset.

24.4 Isolation Forest

Isolation Forest, or iForest for short, is a tree-based anomaly detection algorithm.

... Isolation Forest (iForest) which detects anomalies purely based on the concept of isolation without employing any distance or density measure

— *Isolation-Based Anomaly Detection*, 2012.

It is based on modeling the normal data in such a way to isolate anomalies that are both few in number and different in the feature space.

... our proposed method takes advantage of two anomalies' quantitative properties: i) they are the minority consisting of fewer instances and ii) they have attribute-values that are very different from those of normal instances.

— *Isolation Forest*, 2008.

Tree structures are created to isolate anomalies. The result is that isolated examples have a relatively short depth in the trees, whereas normal data is less isolated and has a greater depth in the trees.

... a tree structure can be constructed effectively to isolate every single instance. Because of their susceptibility to isolation, anomalies are isolated closer to the root of the tree; whereas normal points are isolated at the deeper end of the tree.

— *Isolation Forest*, 2008.

The scikit-learn library provides an implementation of Isolation Forest in the `IsolationForest` class. Perhaps the most important hyperparameters of the model are the `n_estimators` argument that sets the number of trees to create and the `contamination` argument, which is used to help define the number of outliers in the dataset. We know the contamination is about 0.01 percent positive cases to negative cases, so we can set the `contamination` argument to be 0.01.

```
...
# define outlier detection model
model = IsolationForest(contamination=0.01)
```

Listing 24.10: Example of configuring the isolation forest model.

The model is probably best trained on examples that exclude outliers. In this case, we fit the model on the input features for examples from the majority class only.

```
...
# fit on majority class
trainX = trainX[trainy==0]
model.fit(trainX)
```

Listing 24.11: Example of fitting the isolation forest model.

Like one-class SVM, the model will predict an inlier with a label of +1 and an outlier with a label of -1, therefore, the labels of the test set must be changed before evaluating the predictions. Tying this together, the complete example is listed below.

```

# isolation forest for imbalanced classification
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import f1_score
from sklearn.ensemble import IsolationForest
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.999], flip_y=0, random_state=4)
# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2,
    stratify=y)
# define outlier detection model
model = IsolationForest(contamination=0.01)
# fit on majority class
trainX = trainX[trainy==0]
model.fit(trainX)
# detect outliers in the test set
yhat = model.predict(testX)
# mark inliers 1, outliers -1
testy[testy == 1] = -1
testy[testy == 0] = 1
# calculate score
score = f1_score(testy, yhat, pos_label=-1)
print('F-measure: %.3f' % score)

```

Listing 24.12: Example of evaluating the isolation forest model on the severely imbalanced classification dataset.

Running the example fits the isolation forest model on the training dataset in an unsupervised manner, then classifies examples in the test set as inliers and outliers and scores the result.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, an F-measure of 0.154 is achieved.

F-measure: 0.154

Listing 24.13: Example output from evaluating the isolation forest model on the severely imbalanced classification dataset.

Note: the contamination is quite low and may result in many runs with an F-measure of 0.0. To improve the stability of the method on this dataset, try increasing the contamination to 0.05 or even 0.1 and re-run the example.

24.5 Minimum Covariance Determinant

If the input variables have a Gaussian distribution, then simple statistical methods can be used to detect outliers. For example, if the dataset has two input variables and both are Gaussian, then the feature space forms a multi-dimensional Gaussian and knowledge of this distribution can be used to identify values far from the distribution.

This approach can be generalized by defining a hypersphere (ellipsoid) that covers the normal data, and data that falls outside this shape is considered an outlier. An efficient implementation

of this technique for multivariate data is known as the Minimum Covariance Determinant, or MCD for short. It is unusual to have such well-behaved data, but if this is the case for your dataset, or you can use power transforms to make the variables Gaussian, then this approach might be appropriate.

The Minimum Covariance Determinant (MCD) method is a highly robust estimator of multivariate location and scatter, for which a fast algorithm is available. [...] It also serves as a convenient and efficient tool for outlier detection.

— *Minimum Covariance Determinant and Extensions*, 2017.

The scikit-learn library provides access to this method via the `EllipticEnvelope` class. It provides the `contamination` argument that defines the expected ratio of outliers to be observed in practice. We know that this is 0.01 percent in our synthetic dataset, so we can set it accordingly.

```
...
# define outlier detection model
model = EllipticEnvelope(contamination=0.01)
```

Listing 24.14: Example of configuring the elliptic envelope model.

The model can be fit on the input data from the majority class only in order to estimate the distribution of *normal* data in an unsupervised manner.

```
...
# fit on majority class
trainX = trainX[trainy==0]
model.fit(trainX)
```

Listing 24.15: Example of fitting the elliptic envelope model.

The model will then be used to classify new examples as either normal (+1) or outliers (-1).

```
...
# detect outliers in the test set
yhat = model.predict(testX)
```

Listing 24.16: Example of making a prediction with the elliptic envelope model.

Tying this together, the complete example of using the elliptic envelope outlier detection model for imbalanced classification on our synthetic binary classification dataset is listed below.

```
# elliptic envelope for imbalanced classification
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import f1_score
from sklearn.covariance import EllipticEnvelope
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.999], flip_y=0, random_state=4)
# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2,
    stratify=y)
# define outlier detection model
model = EllipticEnvelope(contamination=0.01)
```

```
# fit on majority class
trainX = trainX[trainy==0]
model.fit(trainX)
# detect outliers in the test set
yhat = model.predict(testX)
# mark inliers 1, outliers -1
testy[testy == 1] = -1
testy[testy == 0] = 1
# calculate score
score = f1_score(testy, yhat, pos_label=-1)
print('F-measure: %.3f' % score)
```

Listing 24.17: Example of evaluating the elliptic envelope model on the severely imbalanced classification dataset.

Running the example fits the elliptic envelope model on the training dataset in an unsupervised manner, then classifies examples in the test set as inliers and outliers and scores the result.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, an F-measure of 0.157 is achieved.

```
F-measure: 0.157
```

Listing 24.18: Example output from evaluating the elliptic envelope model on the severely imbalanced classification dataset.

24.6 Local Outlier Factor

A simple approach to identifying outliers is to locate those examples that are far from the other examples in the feature space. This can work well for feature spaces with low dimensionality (few features), although it can become less reliable as the number of features is increased, referred to as the curse of dimensionality.

The local outlier factor, or LOF for short, is a technique that attempts to harness the idea of nearest neighbors for outlier detection. Each example is assigned a scoring of how isolated or how likely it is to be outliers based on the size of its local neighborhood. Those examples with the largest score are more likely to be outliers.

We introduce a local outlier (LOF) for each object in the dataset, indicating its degree of outlier-ness.

— LOF: *Identifying Density-based Local Outliers*, 2000.

The scikit-learn library provides an implementation of this approach in the LocalOutlierFactor class. The model can be defined and requires that the expected percentage of outliers in the dataset be indicated, such as 0.01 percent in the case of our synthetic dataset.

```
...
# define outlier detection model
model = LocalOutlierFactor(contamination=0.01)
```

Listing 24.19: Example of configuring the LOF model.

The model is not fit. Instead, a *normal* dataset is used as the basis for identifying outliers in new data via a call to `fit_predict()`. To use this model to identify outliers in our test dataset, we must first prepare the training dataset to only have input examples from the majority class.

```
...
# get examples for just the majority class
trainX = trainX[trainy==0]
```

Listing 24.20: Example of retrieving the examples from the majority class.

Next, we can concatenate these examples with the input examples from the test dataset.

```
...
# create one large dataset
composite = vstack((trainX, testX))
```

Listing 24.21: Example of creating the composite dataset for prediction.

We can then make a prediction by calling `fit_predict()` and retrieve only those labels for the examples in the test set.

```
...
# make prediction on composite dataset
yhat = model.fit_predict(composite)
# get just the predictions on the test set
yhat = yhat[len(trainX):]
```

Listing 24.22: Example of making a prediction with the LOF model.

To make things easier, we can wrap this up into a new function with the name `lof_predict()` listed below.

```
# make a prediction with a lof model
def lof_predict(model, trainX, testX):
    # create one large dataset
    composite = vstack((trainX, testX))
    # make prediction on composite dataset
    yhat = model.fit_predict(composite)
    # return just the predictions on the test set
    return yhat[len(trainX):]
```

Listing 24.23: Example of a function for making a prediction with the LOF model.

The predicted labels will be +1 for normal and -1 for outliers, like the other outlier detection algorithms in scikit-learn. Tying this together, the complete example of using the LOF outlier detection algorithm for classification with a skewed class distribution is listed below.

```
# local outlier factor for imbalanced classification
from numpy import vstack
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import f1_score
```

```

from sklearn.neighbors import LocalOutlierFactor

# make a prediction with a lof model
def lof_predict(model, trainX, testX):
    # create one large dataset
    composite = vstack((trainX, testX))
    # make prediction on composite dataset
    yhat = model.fit_predict(composite)
    # return just the predictions on the test set
    return yhat[len(trainX):]

# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.999], flip_y=0, random_state=4)
# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2,
    stratify=y)
# define outlier detection model
model = LocalOutlierFactor(contamination=0.01)
# get examples for just the majority class
trainX = trainX[trainy==0]
# detect outliers in the test set
yhat = lof_predict(model, trainX, testX)
# mark inliers 1, outliers -1
testy[testy == 1] = -1
testy[testy == 0] = 1
# calculate score
score = f1_score(testy, yhat, pos_label=-1)
print('F-measure: %.3f' % score)

```

Listing 24.24: Example of evaluating the LOF model on the severely imbalanced classification dataset.

Running the example uses the local outlier factor model with the training dataset in an unsupervised manner to classify examples in the test set as inliers and outliers, then scores the result.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, an F-measure of 0.138 is achieved.

F-measure: 0.138

Listing 24.25: Example output from evaluating the LOF model on the severely imbalanced classification dataset.

24.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

24.7.1 Papers

- *Estimating the Support of a High-Dimensional Distribution*, 2001.
<https://dl.acm.org/citation.cfm?id=1119749>
- *Isolation Forest*, 2008.
<https://ieeexplore.ieee.org/abstract/document/4781136>
- *Isolation-Based Anomaly Detection*, 2012.
<https://dl.acm.org/citation.cfm?id=2133363>
- *A Fast Algorithm for the Minimum Covariance Determinant Estimator*, 2012.
<https://amstat.tandfonline.com/doi/abs/10.1080/00401706.1999.10485670>
- *Minimum Covariance Determinant and Extensions*, 2017.
<https://arxiv.org/abs/1709.07045>
- *LOF: Identifying Density-based Local Outliers*, 2000.
<https://dl.acm.org/citation.cfm?id=335388>

24.7.2 Books

- *Learning from Imbalanced Data Sets*, 2018.
<https://amzn.to/307Xlva>
- *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.
<https://amzn.to/32K9K6d>

24.7.3 APIs

- Novelty and Outlier Detection, scikit-learn API.
https://scikit-learn.org/stable/modules/outlier_detection.html
- sklearn.svm.OneClassSVM API.
<https://scikit-learn.org/stable/modules/generated/sklearn.svm.OneClassSVM.html>
- sklearn.ensemble.IsolationForest API.
<https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html>
- sklearn.covariance.EllipticEnvelope API.
<https://scikit-learn.org/stable/modules/generated/sklearn.covariance.EllipticEnvelope.html>
- sklearn.neighbors.LocalOutlierFactor API.
<https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.LocalOutlierFactor.html>

24.7.4 Articles

- Outlier, Wikipedia.
<https://en.wikipedia.org/wiki/Outlier>
- Anomaly detection, Wikipedia.
https://en.wikipedia.org/wiki/Anomaly_detection
- One-class classification, Wikipedia.
https://en.wikipedia.org/wiki/One-class_classification

24.8 Summary

In this tutorial, you discovered how to use one-class classification algorithms for datasets with severely skewed class distributions. Specifically, you learned:

- One-class classification is a field of machine learning that provides techniques for outlier and anomaly detection.
- How to adapt one-class classification algorithms for imbalanced classification with a severely skewed class distribution.
- How to fit and evaluate one-class classification algorithms such as SVM, isolation forest, elliptic envelope and local outlier factor.

24.8.1 Next

This was the final tutorial in this Part. In the next Part, you will discover how to work through imbalanced classification projects systematically.

Part VII

Projects

Chapter 25

Framework for Imbalanced Classification Projects

Classification predictive modeling problems involve predicting a class label for a given set of inputs. It is a challenging problem in general, especially if little is known about the dataset, as there are tens, if not hundreds, of machine learning algorithms to choose from. The problem is made significantly more difficult if the distribution of examples across the classes is imbalanced. This requires the use of specialized methods to either change the dataset or change the learning algorithm to handle the skewed class distribution.

A common way to deal with the overwhelm on a new classification project is to use a favorite machine learning algorithm like Random Forest or SMOTE. Another common approach is to scour the research literature for descriptions of vaguely similar problems and attempt to re-implement the algorithms and configurations that are described.

These approaches can be effective, although they are hit-or-miss and time-consuming respectively. Instead, the shortest path to a good result on a new classification task is to systematically evaluate a suite of machine learning algorithms in order to discover what works well, then double down on what works well. This approach can also be used for imbalanced classification problems, tailored for the range of data sampling, cost-sensitive, and one-class classification algorithms that one may choose from. In this tutorial, you will discover a systematic framework for working through an imbalanced classification dataset. After completing this tutorial, you will know:

- The challenge of choosing an algorithm for imbalanced classification.
- A high-level framework for systematically working through an imbalanced classification project.
- Specific algorithm suggestions to try at each step of an imbalanced classification project.

Let's get started.

25.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. What Algorithm To Use?

2. Use a Systematic Framework
3. Detailed Framework for Imbalanced Classification

25.2 What Algorithm To Use?

You're handed or acquire an imbalanced classification dataset. Now what? There are so many machine learning algorithms to choose from, let alone techniques specifically designed for imbalanced classification. **Which algorithms do you use? How do you choose?**

This is the challenge faced at the beginning of each new imbalanced classification project. It is this challenge that makes applied machine learning both thrilling and terrifying. There are perhaps two common ways to solve this problem:

- Use a favorite algorithm.
- Use what has worked previously.

One approach might be to select a favorite algorithm and start tuning the hyperparameters. This is a fast approach to a solution but is only effective if your favorite algorithm just happens to be the best solution for your specific dataset. Another approach might be to review the literature and see what techniques have been used on datasets like yours. This can be effective if many people have studied and reported results on similar datasets.

In practice, this is rarely the case, and research publications are biased towards showing promise for a pet algorithm rather than presenting an honest comparison of methods. At best, literature can be used for ideas for techniques to try. Instead, if little is known about the problem, then the shortest path to a *good* result is to systematically test a suite of different algorithms on your dataset.

25.3 Use a Systematic Framework

Consider a balanced classification task. You're faced with the same challenge of selecting which algorithms to use to address your dataset. There are many solutions to this problem, but perhaps the most robust is to systematically test a suite of algorithms and use empirical results to choose. Biases like *my favorite algorithm* or *what has worked in the past* can feed ideas into the study, but can lead you astray if relied upon. Instead, you need to let the results from systematic empirical experiments to tell you what algorithm is good or best for your imbalanced classification dataset.

Once you have a dataset, the process involves the three steps of (1) selecting a metric by which to evaluate candidate models, (2) testing a suite of algorithms, and (3) tuning the best performing models. This may not be the only approach; it is just the simplest reliable process to get you from *I have a new dataset* to *I have good results* very quickly. This process can be summarized as follows:

1. Select a Metric
2. Spot-Check Algorithms

3. Hyperparameter Tuning

Spot-checking algorithms is a little more involved as many algorithms require specialized data preparation such as scaling, removal of outliers, and more. Also, evaluating candidate algorithms requires the careful design of a test harness, often involving the use of k -fold cross-validation to estimate the performance of a given model on unseen data. We can use this simple process for imbalanced classification.

It is still important to spot-check standard machine learning algorithms on imbalanced classification. Standard algorithms often do not perform well when the class distribution is imbalanced. Nevertheless, testing them first provides a baseline in performance by which more specialized models can be compared and must out-perform. It is also still important to tune the hyperparameters of well-performing algorithms. This includes the hyperparameters of models specifically designed for imbalanced classification. Therefore, we can use the same three-step procedure and insert an additional step to evaluate imbalanced classification algorithms. We can summarize this process as follows:

- Select a Metric
- Spot-Check Algorithms
- *Spot-Check Imbalanced Algorithms*
- Hyperparameter Tuning

This provides a high-level systematic framework to work through an imbalanced classification problem. Nevertheless, there are many imbalanced algorithms to choose from, let alone many different standard machine learning algorithms to choose from. We require a similar low-level systematic framework for each step.

25.4 Detailed Framework for Imbalanced Classification

We can develop a similar low-level framework to systematically work through each step of an imbalanced classification project. From selecting a metric to hyperparameter tuning.

25.4.1 Select a Metric

Selecting a metric might be the most important step in the project (for more on choosing a performance metric, see Chapter 4). The metric is the measuring stick by which all models are evaluated and compared. The choice of the wrong metric can mean choosing the wrong algorithm. That is, a model that solves a different problem from the problem you actually want solved. The metric must capture those details about a model or its predictions that are most important to the project or project stakeholders.

This is hard, as there are many metrics to choose from and often project stakeholders are not sure what they want. There may also be multiple ways to frame the problem, and it may be beneficial to explore a few different framings and, in turn, different metrics to see what makes sense to stakeholders. First, you must decide whether you want to predict probabilities or crisp class labels. Recall that for binary imbalanced classification tasks, the majority class is

normal, called the *negative class*, and the minority class is the exception, called the *positive class*. Probabilities capture the uncertainty of the prediction, whereas crisp class labels can be used immediately.

- **Probabilities:** Predict the probability of class membership for each example.
- **Class Labels:** Predict a crisp class label for each example.

Predict Probabilities

If probabilities are intended to be used directly, then a good metric might be the Brier Score and the Brier Skill score. Alternately, you may want to predict probabilities and allow the user to map them to crisp class labels themselves via a user-selected threshold. In this case, a measure can be chosen that summarizes the performance of the model across the range of possible thresholds.

If the positive class is the most important, then the precision-recall curve and area under curve (PR AUC) can be used. This will optimize both precision and recall across all thresholds. Alternately, if both classes are equally important, the ROC Curve and area under curve (ROC AUC) can be used. This will maximize the true positive rate and minimize the false positive rate.

Predict Class Labels

If class labels are required and both classes are equally important, a good default metric is classification accuracy. This only makes sense if the majority class is less than about 80 percent off the data. A majority class that has a greater than 80 percent or 90 percent skew will swamp the accuracy metric and it will lose its meaning for comparing algorithms.

If the class distribution is severely skewed, then the G-mean metric can be used that will optimize the sensitivity and specificity metrics. If the positive class is more important, then variations of the F-measure can be used that optimize the precision and recall. If both false positive and false negatives are equally important, then F1 can be used. If false negatives are more costly, then the F2-measure can be used, otherwise, if false positives are more costly, then the F0.5-measure can be used.

Framework for Choosing a Metric

These are just heuristics but provide a useful starting point if you feel lost choosing a metric for your imbalanced classification task. We can summarize these heuristics into a framework as follows (taken from Chapter 4):

- Are you predicting probabilities?
 - Do you need class labels?
 - * Is the positive class more important?
 - Use Precision-Recall AUC
 - * Are both classes important?
 - Use ROC AUC

- Do you need probabilities?
 - * Use Brier Score and Brier Skill Score
- Are you predicting class labels?
 - Is the positive class more important?
 - * Are False Negatives and False Positives Equally Important?
 - Use F1-Measure
 - * Are False Negatives More Important?
 - Use F2-measure
 - * Are False Positives More Important?
 - Use F0.5-measure
 - Are both classes important?
 - * Do you have < 80%-90% Examples for the Majority Class?
 - Use Accuracy
 - * Do you have > 80%-90% Examples for the Majority Class?
 - Use G-mean

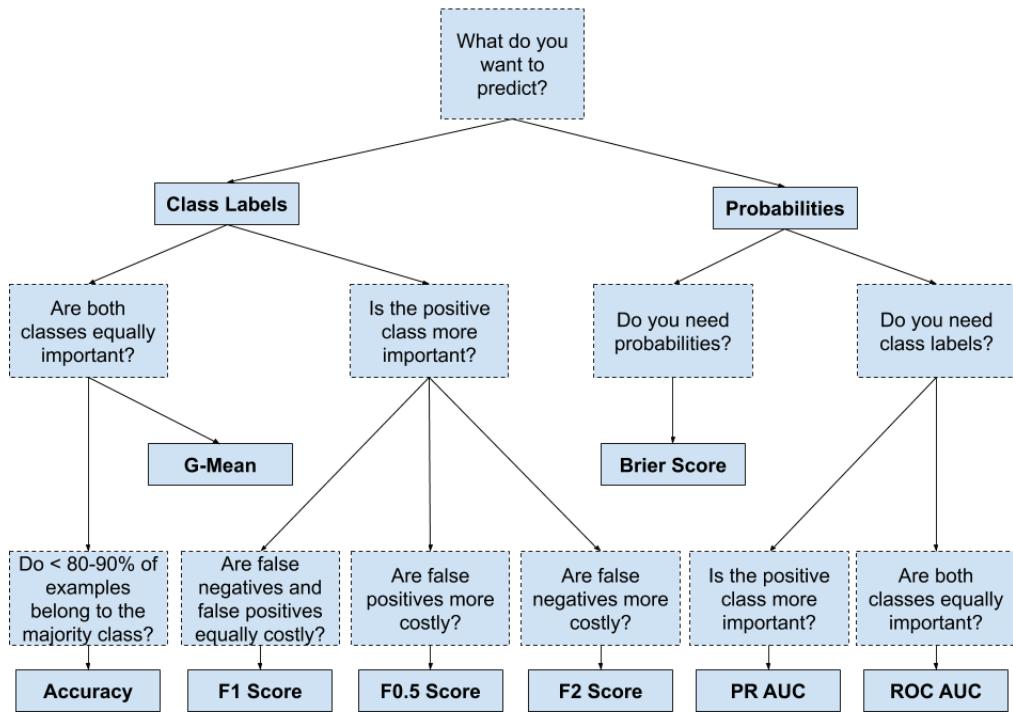


Figure 25.1: How to Choose a Metric for Imbalanced Classification.

Once a metric has been chosen, you can start evaluating machine learning algorithms.

25.4.2 Spot-Check Algorithms

Spot-checking machine learning algorithms means evaluating a suite of different types of algorithms with minimal hyperparameter tuning. Specifically, it means giving each algorithm a good chance to learn about the problem, including performing any required data preparation expected by the algorithm and using best-practice configuration options or defaults. The objective is to quickly test a range of standard machine learning algorithms and provide a baseline in performance to which techniques specialized for imbalanced classification must be compared and outperform in order to be considered skillful. The idea here is that there is little point in using fancy imbalanced algorithms if they cannot out-perform so-called unbalanced algorithms.

A robust test harness must be defined. This often involves k -fold cross-validation, often with $k = 10$ as a sensible default. Stratified cross-validation is often required to ensure that each fold has the same class distribution as the original dataset. And the cross-validation procedure is often repeated multiple times, such as 3, 10, or 30 in order to effectively capture a sample of model performance on the dataset, summarized with a mean and standard deviation of the scores. There are perhaps four levels of algorithms to spot-check; they are:

1. Naive Algorithms
2. Linear Algorithms
3. Nonlinear Algorithms
4. Ensemble Algorithms

Naive Algorithms

Firstly, a naive classification must be evaluated. This provides a rock-bottom baseline in performance that any algorithm must overcome in order to have skill on the dataset. Naive means that the algorithm has no logic other than an if-statement or predicting a constant value. The choice of naive algorithm is based on the choice of performance metric.

For example, a suitable naive algorithm for classification accuracy is to predict the majority class in all cases. A suitable naive algorithm for the Brier Score when evaluating probabilities is to predict the prior probability of each class in the training dataset. A suggested mapping of performance metrics to naive algorithms is as follows:

- **Accuracy:** Predict the majority class (class 0).
- **G-mean:** Predict a uniformly random class.
- **F-measure:** Predict the minority class (class 1).
- **ROC AUC:** Predict a stratified random class.
- **PR ROC:** Predict a stratified random class.
- **Brier Score:** Predict majority class prior.

If you are unsure of the *best* naive algorithm for your metric, perhaps test a few and discover which results in the better performance that you can use as your rock-bottom baseline. Some options include:

- Predict the majority class in all cases.
- Predict the minority class in all cases.
- Predict a uniform randomly selected class.
- Predict a randomly selected class selected with the prior probabilities of each class.
- Predict the class prior probabilities.

Linear Algorithms

Linear algorithms are those that are often drawn from the field of statistics and make strong assumptions about the functional form of the problem. We can refer to them as linear because the output is a linear combination of the inputs, or weighted inputs, although this definition is stretched. You might also refer to these algorithms as probabilistic algorithms as they are often fit under a probabilistic framework. They are often fast to train and often perform very well. Examples of linear algorithms you should consider trying include:

- Logistic Regression
- Linear Discriminant Analysis
- Naive Bayes

Nonlinear Algorithms

Nonlinear algorithms are drawn from the field of machine learning and make few assumptions about the functional form of the problem. We can refer to them as nonlinear because the output is often a nonlinear mapping of inputs to outputs. They often require more data than linear algorithms and are slower to train. Examples of nonlinear algorithms you should consider trying include:

- Decision Tree
- k -Nearest Neighbors
- Artificial Neural Networks
- Support Vector Machine

Ensemble Algorithms

Ensemble algorithms are also drawn from the field of machine learning and combine the predictions from two or more models. There are many ensemble algorithms to choose from, but when spot-checking algorithms, it is a good idea to focus on ensembles of decision tree algorithms, given that they are known to perform so well in practice on a wide range of problems. Examples of ensembles of decision tree algorithms you should consider trying include:

- Bagged Decision Trees
- Random Forest
- Extra Trees
- Stochastic Gradient Boosting

Framework for Spot-Checking Machine Learning Algorithms

We can summarize these suggestions into a framework for testing machine learning algorithms on a dataset.

- Naive Algorithms
 - Majority Class
 - Minority Class
 - Class Priors
- Linear Algorithms
 - Logistic Regression
 - Linear Discriminant Analysis
 - Naive Bayes
- Nonlinear Algorithms
 - Decision Tree
 - k -Nearest Neighbors
 - Artificial Neural Networks
 - Support Vector Machine
- Ensemble Algorithms
 - Bagged Decision Trees
 - Random Forest
 - Extra Trees
 - Stochastic Gradient Boosting

The order of the steps is probably not flexible. Think of the order of algorithms as increasing in complexity, and in turn, capability. The order of algorithms within each step is flexible and the list of algorithms is not complete, and probably never could be given the vast number of algorithms available. Limiting the algorithms tested to a subset of the most common or most widely used is a good start. Using data-preparation recommendations and hyperparameter defaults is also a good start. The figure below summarizes this step of the framework.

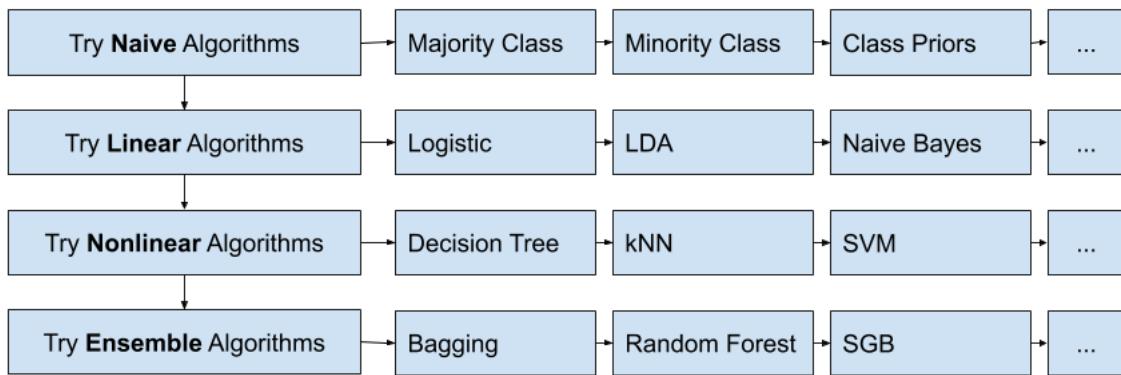


Figure 25.2: How to Spot-Check Machine Learning Algorithms.

25.4.3 Spot-Check Imbalanced Algorithms

Spot-checking imbalanced algorithms is much like spot-checking balanced machine learning algorithms. The objective is to quickly test a large number of techniques in order to discover what shows promise so that you can focus more attention on it later during hyperparameter tuning. The spot-checking performed in the previous section provides both naive and modestly skillful models by which all imbalanced techniques can be compared. This allows you to focus on these methods that truly show promise on the problem, rather than getting excited about results that only appear effective compared only to other imbalanced classification techniques (which is an easy trap to fall into). There are perhaps four types of imbalanced classification techniques to spot-check:

1. Data Sampling Algorithms
2. Cost-Sensitive Algorithms
3. One-Class Algorithms
4. Probability Tuning Algorithms

Data Sampling Algorithms

Data sampling algorithms change the composition of the training dataset to improve the performance of a standard machine learning algorithm on an imbalanced classification problem (for more on data sampling, see Chapter 10). There are perhaps three main types of data sampling techniques; they are:

- Data Oversampling.

- Data Undersampling.
- Combined Oversampling and Undersampling.

Data oversampling involves duplicating examples of the minority class or synthesizing new examples from the minority class from existing examples. Perhaps the most popular methods is SMOTE and variations such as Borderline SMOTE. Perhaps the most important hyperparameter to tune is the amount of oversampling to perform. Examples of data oversampling methods include:

- Random Oversampling
- SMOTE
- Borderline SMOTE
- SVM SMOTE
- k -Means SMOTE
- ADASYN

Undersampling involves deleting examples from the majority class, such as randomly or using an algorithm to carefully choose which examples to delete. Popular editing algorithms include the edited nearest neighbors and Tomek links.

Examples of data undersampling methods include:

- Random Undersampling
- Condensed Nearest Neighbor
- Tomek Links
- Edited Nearest Neighbors
- Neighborhood Cleaning Rule
- One-Sided Selection

Almost any oversampling method can be combined with almost any undersampling technique. Therefore, it may be beneficial to test a suite of different combinations of oversampling and undersampling techniques. Examples of popular combinations of over and undersampling include:

- SMOTE and Random Undersampling
- SMOTE and Tomek Links
- SMOTE and Edited Nearest Neighbors

Data sampling algorithms may perform differently depending on the choice of machine learning algorithm. As such, it may be beneficial to test a suite of standard machine learning algorithms, such as all or a subset of those algorithms used when spot-checking in the previous section. Additionally, most data sampling algorithms make use of the k -nearest neighbor algorithm internally. This algorithm is very sensitive to the data types and scale of input variables. As such, it may be important to at least normalize input variables that have differing scales prior to testing the methods, and perhaps using specialized methods if some input variables are categorical instead of numerical.

Cost-Sensitive Algorithms

Cost-sensitive algorithms are modified versions of machine learning algorithms designed to take the differing costs of misclassification into account when fitting the model on the training dataset (for more on cost-sensitive learning, see Chapter 15). These algorithms can be effective when used on imbalanced classification, where the cost of misclassification is configured to be inversely proportional to the distribution of examples in the training dataset.

There are many cost-sensitive algorithms to choose from, although it might be practical to test a range of cost-sensitive versions of linear, nonlinear, and ensemble algorithms. Some examples of machine learning algorithms that can be configured using cost-sensitive training include:

- Logistic Regression
- Decision Trees
- Support Vector Machines
- Artificial Neural Networks
- Bagged Decision Trees
- Random Forest
- Stochastic Gradient Boosting

One-Class Algorithms

Algorithms used for outlier detection and anomaly detection can be used for classification tasks. Although unusual, when used in this way, they are often referred to as one-class classification algorithms (for more on one-class algorithms, see Chapter 24). In some cases, one-class classification algorithms can be very effective, such as when there is a severe class imbalance with very few examples of the positive class. Examples of one-class classification algorithms to try include:

- One-Class Support Vector Machines
- Isolation Forests
- Minimum Covariance Determinant
- Local Outlier Factor

Probability Tuning Algorithms

Predicted probabilities can be improved in two ways; they are:

- Calibrating Probabilities.
- Tuning the Classification Threshold.

Calibrating Probabilities Some algorithms are fit using a probabilistic framework and, in turn, have calibrated probabilities. This means that when 100 examples are predicted to have the positive class label with a probability of 80 percent, then the algorithm will predict the correct class label 80 percent of the time. Calibrated probabilities are required from a model to be considered skillful on a binary classification task when probabilities are either required as the output or used to evaluate the model (e.g. ROC AUC or PR AUC). Some examples of machine learning algorithms that predict calibrated probabilities are as follows:

- Logistic Regression
- Linear Discriminant Analysis
- Naive Bayes
- Artificial Neural Networks

Most nonlinear algorithms do not predict calibrated probabilities, therefore algorithms can be used to post-process the predicted probabilities in order to calibrate them. Therefore, when probabilities are required directly or are used to evaluate a model, and nonlinear algorithms are being used, it is important to calibrate the predicted probabilities (for more on calibrating probabilities, see Chapter 22). Some examples of probability calibration algorithms to try include:

- Platt Scaling
- Isotonic Regression

Tuning the Classification Threshold Some algorithms are designed to natively predict probabilities that later must be mapped to crisp class labels. This is the case if class labels are required as output for the problem, or the model is evaluated using class labels. Examples of probabilistic machine learning algorithms that predict a calibrated probability by default include:

- Logistic Regression
- Linear Discriminant Analysis
- Naive Bayes
- Artificial Neural Networks

Probabilities are mapped to class labels using a threshold probability value (for more on probability thresholds, see Chapter 21). All probabilities below the threshold are mapped to class 0, and all probabilities equal-to or above the threshold are mapped to class 1. The default threshold is 0.5 for balanced data, although different thresholds can be used that will dramatically impact the class labels and, in turn, the performance of a machine learning model that natively predicts probabilities. As such, if probabilistic algorithms are used that natively predict a probability and class labels are required as output or used to evaluate models, it is a good idea to try tuning the classification threshold.

Framework for Spot-Checking Imbalanced Algorithms

We can summarize these suggestions into a framework for testing imbalanced machine learning algorithms on a dataset.

- Data Sampling Algorithms
 - Data Oversampling
 - * Random Oversampling
 - * SMOTE
 - * Borderline SMOTE
 - * SVM SMOTE
 - * k -Means SMOTE
 - * ADASYN
 - Data Undersampling
 - * Random Undersampling
 - * Condensed Nearest Neighbor
 - * Tomek Links
 - * Edited Nearest Neighbors
 - * Neighborhood Cleaning Rule
 - * One Sided Selection
 - Combined Oversampling and Undersampling
 - * SMOTE and Random Undersampling
 - * SMOTE and Tomek Links
 - * SMOTE and Edited Nearest Neighbors
- Cost-Sensitive Algorithms
 - Logistic Regression
 - Decision Trees
 - Support Vector Machines
 - Artificial Neural Networks
 - Bagged Decision Trees
 - Random Forest

- Stochastic Gradient Boosting
- One-Class Algorithms
 - One-Class Support Vector Machines
 - Isolation Forests
 - Minimum Covariance Determinant
 - Local Outlier Factor
- Probability Tuning Algorithms
 - Calibrating Probabilities
 - * Platt Scaling
 - * Isotonic Regression
 - Tuning the Classification Threshold

The order of the steps is flexible, and the order of algorithms within each step is also flexible, and the list of algorithms is not complete. The structure is designed to get you thinking systematically about what algorithm to evaluate. The figure below summarizes the framework.

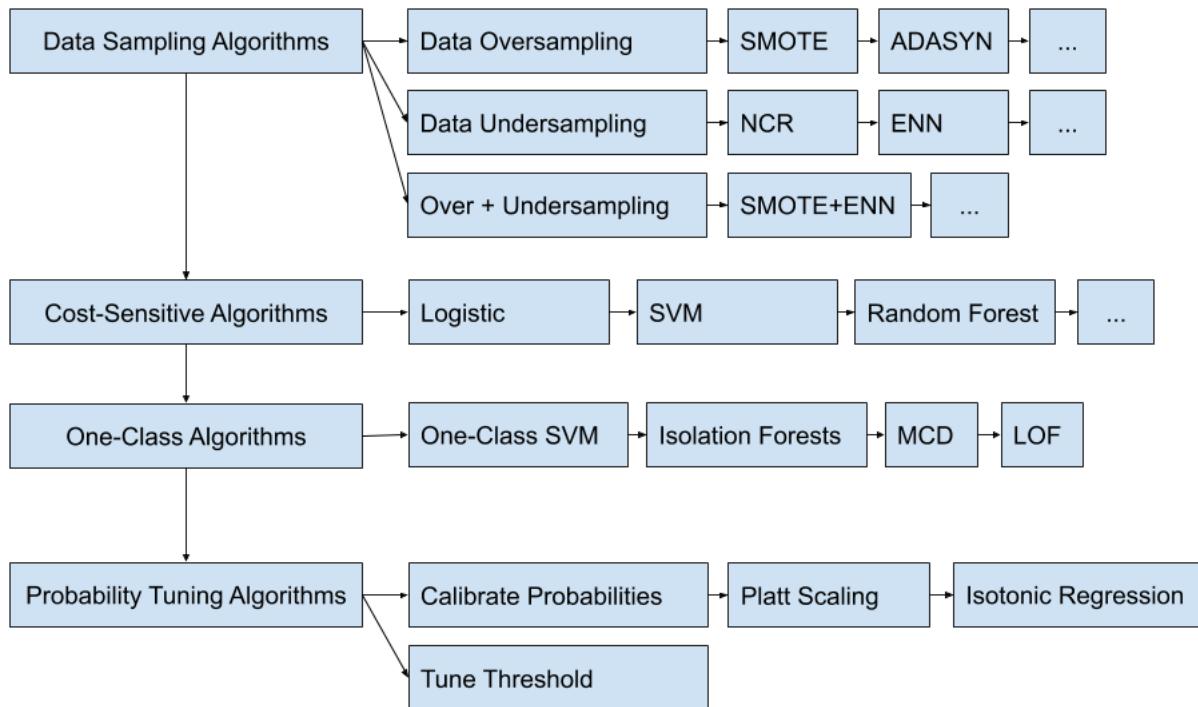


Figure 25.3: How to Spot-Check Imbalanced Machine Learning Algorithms.

25.4.4 Hyperparameter Tuning

After spot-checking machine learning algorithms and imbalanced algorithms, you will have some idea of what works and what does not on your specific dataset. The simplest approach to

hyperparameter tuning is to select the top five or 10 algorithms or algorithm combinations that performed well and tune the hyperparameters for each. There are three popular hyperparameter tuning algorithms that you may choose from:

- Random Search
- Grid Search
- Bayesian Optimization

A good default is grid search if you know what hyperparameter values to try, otherwise, random search should be used. Bayesian optimization should be used if possible but can be more challenging to set up and run. Tuning the best performing methods is a good start, but not the only approach. There may be some standard machine learning algorithms that perform well, but do not perform as well when used with data sampling or probability calibration. These algorithms could be tuned in concert with their imbalanced-classification augmentations to see if better performance can be achieved.

Additionally, there may be imbalanced-classification algorithms, such as a data sampling method that results in a dramatic lift in performance for one or more algorithms. These algorithms themselves may provide an interesting basis for further tuning to see if additional lifts in performance can be achieved.

25.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

25.5.1 Books

- *Learning from Imbalanced Data Sets*, 2018.
<https://amzn.to/307Xlva>
- *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.
<https://amzn.to/32K9K6d>

25.6 Summary

In this tutorial, you discovered a systematic framework for working through an imbalanced classification dataset. Specifically, you learned:

- The challenge of choosing an algorithm for imbalanced classification.
- A high-level framework for systematically working through an imbalanced classification project.
- Specific algorithm suggestions to try at each step of an imbalanced classification project.

25.6.1 Next

In the next tutorial, you will discover how to systematically work through the Haberman breast cancer survival imbalanced classification dataset.

Chapter 26

Project: Haberman Breast Cancer Classification

Developing a probabilistic model is challenging in general, although it is made more so when there is skew in the distribution of cases, referred to as an imbalanced dataset. The Haberman Dataset describes the five year or greater survival of breast cancer patient patients in the 1950s and 1960s and mostly contains patients that survive. This standard machine learning dataset can be used as the basis of developing a probabilistic model that predicts the probability of survival of a patient given a few details of their case.

Given the skewed distribution in cases in the dataset, careful attention must be paid to both the choice of predictive models to ensure that calibrated probabilities are predicted, and to the choice of model evaluation to ensure that the models are selected based on the skill of their predicted probabilities rather than crisp survival vs. non-survival class labels. In this tutorial, you will discover how to develop a model to predict the probability of patient survival on an imbalanced dataset. After completing this tutorial, you will know:

- How to load and explore the dataset and generate ideas for data preparation and model selection.
- How to evaluate a suite of probabilistic models and improve their performance with appropriate data preparation.
- How to fit a final model and use it to predict probabilities for specific cases.

Let's get started.

26.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Haberman Breast Cancer Survival Dataset
2. Explore the Dataset
3. Model Test and Baseline Result
4. Evaluate Probabilistic Models
5. Make Prediction on New Data

26.2 Haberman Breast Cancer Survival Dataset

In this project, we will use a small breast cancer survival dataset, referred to generally as the *Haberman Dataset*. The dataset describes breast cancer patient data and the outcome is patient survival. Specifically whether the patient survived for five years or longer, or whether the patient did not survive. This is a standard dataset used in the study of imbalanced classification. According to the dataset description, the breast cancer surgery operations were conducted between 1958 and 1970 at the University of Chicago's Billings Hospital. There are 306 examples in the dataset, and there are 3 input variables; they are:

- The age of the patient at the time of the operation.
- The two-digit year of the operation.
- The number of *positive axillary nodes* detected, a measure of a cancer has spread.

As such, we have no control over the selection of cases that make up the dataset or features to use in those cases, other than what is available in the dataset. Although the dataset describes breast cancer patient survival, given the small dataset size and the fact the data is based on breast cancer diagnosis and operations many decades ago, any models built on this dataset are not expected to generalize.

To be crystal clear, we are not *solving breast cancer*. We are exploring a standard imbalanced classification dataset. We will choose to frame this dataset as the prediction of a probability of patient survival. That is: Given patient breast cancer surgery details, what is the probability of survival of the patient to five years or more?

This will provide the basis for exploring probabilistic algorithms that can predict a probability instead of a class label and metrics for evaluating models that predict probabilities instead of class labels. Next, let's take a closer look at the data.

26.3 Explore the Dataset

First, download the dataset and save it in your current working directory with the name `haberman.csv`.

- Download Haberman's Survival Data (`haberman.csv`). ¹

Review the contents of the file. The first few lines of the file should look as follows:

```
30,64,1,1  
30,62,3,1  
30,65,0,1  
31,59,2,1  
31,65,4,1  
33,58,10,1  
33,60,0,1  
34,59,0,2  
34,66,9,2  
34,58,30,1
```

¹<https://raw.githubusercontent.com/jbrownlee/Datasets/master/haberman.csv>

```
...
```

Listing 26.1: Example of rows of data from the Haberman's survival dataset.

We can see that the patients have an age like 30 or 31 (column 1), that operations occurred in years like 64 and 62 for 1964 and 1962 respectively (column 2), and that *axillary nodes* has values like 1 and 0. All values are numeric; specifically, they are integer. We can also see that the class label (column 3) has a value of either 1 for patient survival and 2 for patient non-survival.

Firstly, we can load the CSV dataset and summarize each column using a five-number summary. The dataset can be loaded as a `DataFrame` using the `read_csv()` Pandas function, specifying the location and the names of the columns as there is no header line.

```
...
# define the dataset location
filename = 'haberman.csv'
# define the dataset column names
columns = ['age', 'year', 'nodes', 'class']
# load the csv file as a data frame
dataframe = read_csv(filename, header=None, names=columns)
```

Listing 26.2: Example of loading the Haberman's survival dataset.

We can then call the `describe()` function to create a summary of each column and print the contents of the report. A summary for a column includes useful details like the min and max values, the mean and standard deviation of which are useful if the variable has a Gaussian distribution, and the 25th, 50th, and 75th quartiles, which are useful if the variable does not have a Gaussian distribution.

```
...
# summarize each column
report = dataframe.describe()
print(report)
```

Listing 26.3: Example of summarizing the loaded dataset.

Tying this together, the complete example of loading and summarizing the dataset columns is listed below.

```
# load and summarize the dataset
from pandas import read_csv
# define the dataset location
filename = 'haberman.csv'
# define the dataset column names
columns = ['age', 'year', 'nodes', 'class']
# load the csv file as a data frame
dataframe = read_csv(filename, header=None, names=columns)
# summarize each column
report = dataframe.describe()
print(report)
```

Listing 26.4: Load and summarize the Haberman's survival dataset.

Running the example loads the dataset and reports a summary for each of the three input variables and the output variable. Looking at the age, we can see that the youngest patient was 30 and the oldest was 83; that is quite a range. The mean patient age was about 52 years. If the occurrence of cancer is somewhat random, we might expect the age distribution to be Gaussian.

We can see that all operations were performed between 1958 and 1969. If the number of breast cancer patients is somewhat fixed over time, we might expect this variable to have a uniform distribution. We can see nodes have values between 0 and 52. This might be a cancer diagnostic related to lymphatic nodes.

	age	year	nodes	class
count	306.000000	306.000000	306.000000	306.000000
mean	52.457516	62.852941	4.026144	1.264706
std	10.803452	3.249405	7.189654	0.441899
min	30.000000	58.000000	0.000000	1.000000
25%	44.000000	60.000000	0.000000	1.000000
50%	52.000000	63.000000	1.000000	1.000000
75%	60.750000	65.750000	4.000000	2.000000
max	83.000000	69.000000	52.000000	2.000000

Listing 26.5: Example output from loading and summarizing the Haberman's survival dataset.

All variables are integers. Therefore, it might be helpful to look at each variable as a histogram to get an idea of the variable distribution. This might be helpful in case we choose models later that are sensitive to the data distribution or scale of the data, in which case, we might need to transform or rescale the data. We can create a histogram of each variable in the `DataFrame` by calling the `hist()` function. The complete example is listed below.

```
# create histograms of each variable
from pandas import read_csv
from matplotlib import pyplot
# define the dataset location
filename = 'haberman.csv'
# define the dataset column names
columns = ['age', 'year', 'nodes', 'class']
# load the csv file as a data frame
dataframe = read_csv(filename, header=None, names=columns)
# create a histogram plot of each variable
dataframe.hist()
pyplot.show()
```

Listing 26.6: Create histogram plots for the Haberman's survival dataset.

Running the example creates a histogram for each variable. We can see that age appears to have a Gaussian distribution, as we might have expected. We can also see that year has a uniform distribution, mostly, with an outlier in the first year showing nearly double the number of operations. We can see nodes has an exponential type distribution with perhaps most examples showing 0 nodes, with a long tail of values after that. A transform to un-bunch this distribution might help some models later on. Finally, we can see the two-class values with an unequal class distribution, showing perhaps 2- or 3-times more survival than non-survival cases.

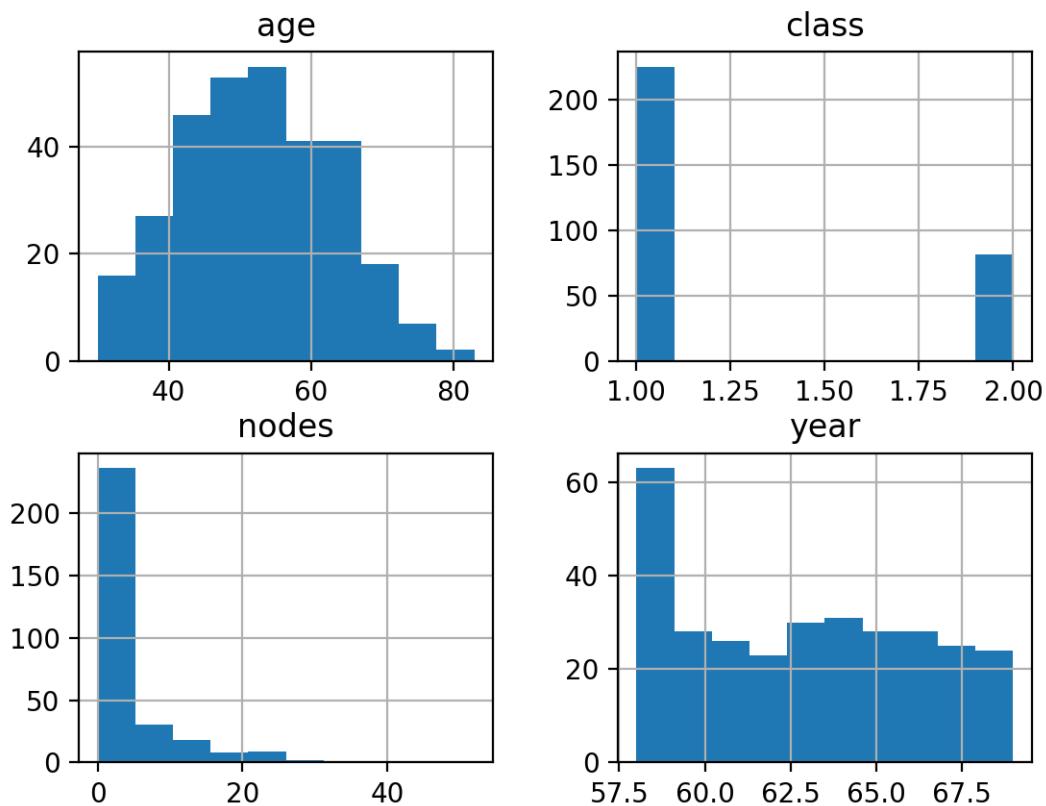


Figure 26.1: Histogram of Each Variable in the Haberman Breast Cancer Survival Dataset.

It may be helpful to know how imbalanced the dataset actually is. We can use the Counter object to count the number of examples in each class, then use those counts to summarize the distribution. The complete example is listed below.

```
# summarize the class ratio
from pandas import read_csv
from collections import Counter
# define the dataset location
filename = 'haberman.csv'
# define the dataset column names
columns = ['age', 'year', 'nodes', 'class']
# load the csv file as a data frame
dataframe = read_csv(filename, header=None, names=columns)
# summarize the class distribution
target = dataframe['class'].values
counter = Counter(target)
for k,v in counter.items():
    per = v / len(target) * 100
    print('Class=%d, Count=%d, Percentage=%.3f%%' % (k, v, per))
```

Listing 26.7: Summarize the class distribution for the Haberman's survival dataset.

Running the example summarizes the class distribution for the dataset. We can see that class 1 for survival has the most examples at 225, or about 74 percent of the dataset. We can

see class 2 for non-survival has fewer examples at 81, or about 26 percent of the dataset. The class distribution is skewed, but it is not severely imbalanced.

```
Class=1, Count=225, Percentage=73.529%
Class=2, Count=81, Percentage=26.471%
```

Listing 26.8: Example output from summarizing the class distribution for the Haberman's survival dataset.

Now that we have reviewed the dataset, let's look at developing a test harness for evaluating candidate models.

26.4 Model Test and Baseline Result

We will evaluate candidate models using repeated stratified k -fold cross-validation. The k -fold cross-validation procedure provides a good general estimate of model performance that is not too optimistically biased, at least compared to a single train-test split. We will use $k = 10$, meaning each fold will contain $\frac{306}{10}$ or about 30 examples.

Stratified means that each fold will contain the same mixture of examples by class, that is about 74 percent to 26 percent survival and non-survival. Repeated means that the evaluation process will be performed multiple times to help avoid fluke results and better capture the variance of the chosen model. We will use three repeats. This means a single model will be fit and evaluated 10×3 (30) times and the mean and standard deviation of these runs will be reported. This can be achieved using the `RepeatedStratifiedKFold` scikit-learn class.

Given that we are interested in predicting a probability of survival, we need a performance metric that evaluates the skill of a model based on the predicted probabilities. In this case, we will use the Brier score that calculates the mean squared error between the predicted probabilities and the expected probabilities (for more on the Brier Score see Chapter 8). This can be calculated using the `brier_score_loss()` scikit-learn function. This score is minimized, with a perfect score of 0.0. We can invert the score to be maximizing by comparing a predicted score to a reference score, showing how much better the model is compared to the reference between 0.0 for the same, to 1.0 with perfect skill. Any models that achieve a score less than 0.0 represents less skill than the reference model. This is called the Brier Skill Score, or BSS for short.

It is customary for an imbalanced dataset to model the minority class as a positive class. In this dataset, the positive class represents non-survival. This means that we will be predicting the probability of non-survival and will need to calculate the complement of the predicted probability in order to get the probability of survival. As such, we can map the 1 class values (survival) to the negative case with a 0 class label, and the 2 class values (non-survival) to the positive case with a class label of 1. This can be achieved using the `LabelEncoder` class. For example, the `load_dataset()` function below will load the dataset, split the variable columns into input and outputs, and then encode the target variable to 0 and 1 values.

```
# load the dataset
def load_dataset(full_path):
    # load the dataset as a numpy array
    data = read_csv(full_path, header=None)
    # retrieve numpy array
    data = data.values
```

```
# split into input and output elements
X, y = data[:, :-1], data[:, -1]
# label encode the target variable to have the classes 0 and 1
y = LabelEncoder().fit_transform(y)
return X, y
```

Listing 26.9: Example a function for loading and preparing the dataset.

Next, we can calculate the Brier skill score for a model. First, we need a Brier score for a reference prediction. A reference prediction for a problem in which we are predicting probabilities is the probability of the positive class label in the dataset. In this case, the positive class label represents non-survival and occurs about 26% in the dataset. Therefore, predicting about 0.26471 represents the worst-case or baseline performance for a predictive model on this dataset. Any model that has a Brier score better than this has some skill, where as any model that has a Brier score lower than this has no skill. The Brier Skill Score captures this important relationship. We can calculate the Brier score for this default prediction strategy automatically for each training set in the k -fold cross-validation process, then use it as a point of comparison for a given model.

```
...
# calculate reference brier score
ref_probs = [0.26471 for _ in range(len(y_true))]
bs_ref = brier_score_loss(y_true, ref_probs)
```

Listing 26.10: Example of calculating the reference Brier score.

The Brier score can then be calculated for the predictions from a model and used in the calculation of the Brier Skill Score. The `brier_skill_score()` function below implements this and calculates the Brier Skill Score for a given set of true labels and predictions on the same test set. Any model that achieves a BSS above 0.0 means it shows skill on this dataset.

```
# calculate brier skill score (BSS)
def brier_skill_score(y_true, y_prob):
    # calculate reference brier score
    pos_prob = count_nonzero(y_true) / len(y_true)
    ref_probs = [pos_prob for _ in range(len(y_true))]
    bs_ref = brier_score_loss(y_true, ref_probs)
    # calculate model brier score
    bs_model = brier_score_loss(y_true, y_prob)
    # calculate skill score
    return 1.0 - (bs_model / bs_ref)
```

Listing 26.11: Example of a function for calculating the Brier Skill Score.

Next, we can make use of the `brier_skill_score()` function to evaluate a model using repeated stratified k -fold cross-validation. To use our custom performance metric, we can use the `make_scorer()` scikit-learn function that takes the name of our custom function and creates a metric that we can use to evaluate models with the scikit-learn API. We will set the `needs_proba` argument to True to ensure that models that are evaluated make predictions using the `predict_proba()` function to ensure they give probabilities instead of class labels.

```
...
# define the model evaluation the metric
metric = make_scorer(brier_skill_score, needs_proba=True)
```

Listing 26.12: Example of a defining a scoring function for the Brier Skill Score.

The `evaluate_model()` function below defines the evaluation procedure with our custom evaluation metric, taking the entire training dataset and model as input, then returns the sample of scores across each fold and each repeat.

```
# evaluate a model
def evaluate_model(X, y, model):
    # define evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # define the model evaluation the metric
    metric = make_scorer(brier_skill_score, needs_proba=True)
    # evaluate model
    scores = cross_val_score(model, X, y, scoring=metric, cv=cv, n_jobs=-1)
    return scores
```

Listing 26.13: Example of a function for evaluating a model.

Finally, we can use our three functions and evaluate a model. First, we can load the dataset and summarize the input and output arrays to confirm they were loaded correctly.

```
...
# define the location of the dataset
full_path = 'haberman.csv'
# load the dataset
X, y = load_dataset(full_path)
# summarize the loaded dataset
print(X.shape, y.shape, Counter(y))
```

Listing 26.14: Example of loading and summarizing the dataset.

In this case, we will evaluate the baseline strategy of predicting the distribution of positive examples in the training set as the probability of each case in the test set. This can be implemented automatically using the `DummyClassifier` class and setting the `strategy` to '`prior`' that will predict the prior probability of each class in the training dataset, which for the positive class we know is about 0.26471.

```
...
# define the reference model
model = DummyClassifier(strategy='prior')
```

Listing 26.15: Example of defining the baseline model.

We can then evaluate the model by calling our `evaluate_model()` function and report the mean and standard deviation of the results.

```
...
# evaluate the model
scores = evaluate_model(X, y, model)
# summarize performance
print('Mean BSS: %.3f (%.3f)' % (mean(scores), std(scores)))
```

Listing 26.16: Example of evaluating the model and summarizing the performance.

Tying this all together, the complete example of evaluating the baseline model on the Haberman breast cancer survival dataset using the Brier Skill Score is listed below. We would expect the baseline model to achieve a BSS of 0.0, e.g. the same as the reference model because it is the reference model.

```
# baseline model and test harness for the haberman dataset
from collections import Counter
from numpy import mean
from numpy import std
from pandas import read_csv
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.metrics import brier_score_loss
from sklearn.metrics import make_scorer
from sklearn.dummy import DummyClassifier

# load the dataset
def load_dataset(full_path):
    # load the dataset as a numpy array
    data = read_csv(full_path, header=None)
    # retrieve numpy array
    data = data.values
    # split into input and output elements
    X, y = data[:, :-1], data[:, -1]
    # label encode the target variable to have the classes 0 and 1
    y = LabelEncoder().fit_transform(y)
    return X, y

# calculate brier skill score (BSS)
def brier_skill_score(y_true, y_prob):
    # calculate reference brier score
    ref_probs = [0.26471 for _ in range(len(y_true))]
    bs_ref = brier_score_loss(y_true, ref_probs)
    # calculate model brier score
    bs_model = brier_score_loss(y_true, y_prob)
    # calculate skill score
    return 1.0 - (bs_model / bs_ref)

# evaluate a model
def evaluate_model(X, y, model):
    # define evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # define the model evaluation metric
    metric = make_scorer(brier_skill_score, needs_proba=True)
    # evaluate model
    scores = cross_val_score(model, X, y, scoring=metric, cv=cv, n_jobs=-1)
    return scores

# define the location of the dataset
full_path = 'haberman.csv'
# load the dataset
X, y = load_dataset(full_path)
# summarize the loaded dataset
print(X.shape, y.shape, Counter(y))
# define the reference model
model = DummyClassifier(strategy='prior')
# evaluate the model
scores = evaluate_model(X, y, model)
# summarize performance
```

```
print('Mean BSS: %.3f (%.3f)' % (mean(scores), std(scores)))
```

Listing 26.17: Calculate the baseline performance for the Haberman’s survival dataset.

Running the example first loads the dataset and reports the number of cases correctly as 306 and the distribution of class labels for the negative and positive cases as we expect. The `DummyClassifier` with our default strategy is then evaluated using repeated stratified k -fold cross-validation and the mean and standard deviation of the Brier Skill Score is reported as 0.0. This is as we expected, as we are using the test harness to evaluate the reference strategy.

```
(306, 3) (306,) Counter({0: 225, 1: 81})  
Mean BSS: -0.000 (0.000)
```

Listing 26.18: Example output from calculating the baseline performance for the Haberman’s survival dataset.

Now that we have a test harness and a baseline in performance, we can begin to evaluate some models on this dataset.

26.5 Evaluate Probabilistic Models

In this section, we will use the test harness developed in the previous section to evaluate a suite of algorithms and then improvements to those algorithms, such as data preparation schemes.

26.5.1 Probabilistic Algorithm Evaluation

We will evaluate a suite of models that are known to be effective at predicting probabilities. Specifically, these are models that are fit under a probabilistic framework and explicitly predict a calibrated probability for each example. A such, this makes them well-suited to this dataset, even with the class imbalance. We will evaluate the following six probabilistic models implemented with the scikit-learn library:

- Logistic Regression (LR)
- Linear Discriminant Analysis (LDA)
- Quadratic Discriminant Analysis (QDA)
- Gaussian Naive Bayes (GNB)
- Multinomial Naive Bayes (MNB)
- Gaussian Process (GPC)

We are interested in directly comparing the results from each of these algorithms. We will compare each algorithm based on the mean score, as well as based on their distribution of scores. We can define a function to create models that we want to evaluate, each with their default configuration or configured as to not produce a warning.

```
# define models to test
def get_models():
    models, names = list(), list()
    # LR
    models.append(LogisticRegression(solver='lbfgs'))
    names.append('LR')
    # LDA
    models.append(LinearDiscriminantAnalysis())
    names.append('LDA')
    # QDA
    models.append(QuadraticDiscriminantAnalysis())
    names.append('QDA')
    # GNB
    models.append(GaussianNB())
    names.append('GNB')
    # MNB
    models.append(MultinomialNB())
    names.append('MNB')
    # GPC
    models.append(GaussianProcessClassifier())
    names.append('GPC')
return models, names
```

Listing 26.19: Example of a function for defining the models to evaluate.

We can then enumerate each model, record a unique name for the model, evaluate it, and report the mean BSS and store the results for the end of the run.

```
...
results = list()
# evaluate each model
for i in range(len(models)):
    # evaluate the model and store results
    scores = evaluate_model(X, y, models[i])
    results.append(scores)
    # summarize and store
    print('>%s %.3f (%.3f)' % (names[i], mean(scores), std(scores)))
```

Listing 26.20: Example of evaluating the defined models and summarizing performance.

At the end of the run, we can then create a box and whisker plot that shows the distribution of results from each algorithm, where the box shows the 25th, 50th, and 75th percentiles of the scores and the triangle shows the mean result. The whiskers of each plot give an idea of the extremes of each distribution.

```
...
# plot the results
pyplot.boxplot(values, labels=names, showmeans=True)
pyplot.show()
```

Listing 26.21: Example of summarizing performance using box and whisker plots.

Tying this together, the complete example is listed below.

```
# compare probabilistic model on the haberman dataset
from numpy import mean
from numpy import std
```

```
from pandas import read_csv
from matplotlib import pyplot
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.metrics import brier_score_loss
from sklearn.metrics import make_scorer
from sklearn.linear_model import LogisticRegression
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.naive_bayes import MultinomialNB
from sklearn.gaussian_process import GaussianProcessClassifier

# load the dataset
def load_dataset(full_path):
    # load the dataset as a numpy array
    data = read_csv(full_path, header=None)
    # retrieve numpy array
    data = data.values
    # split into input and output elements
    X, y = data[:, :-1], data[:, -1]
    # label encode the target variable to have the classes 0 and 1
    y = LabelEncoder().fit_transform(y)
    return X, y

# calculate brier skill score (BSS)
def brier_skill_score(y_true, y_prob):
    # calculate reference brier score
    ref_probs = [0.26471 for _ in range(len(y_true))]
    bs_ref = brier_score_loss(y_true, ref_probs)
    # calculate model brier score
    bs_model = brier_score_loss(y_true, y_prob)
    # calculate skill score
    return 1.0 - (bs_model / bs_ref)

# evaluate a model
def evaluate_model(X, y, model):
    # define evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # define the model evaluation metric
    metric = make_scorer(brier_skill_score, needs_proba=True)
    # evaluate model
    scores = cross_val_score(model, X, y, scoring=metric, cv=cv, n_jobs=-1)
    return scores

# define models to test
def get_models():
    models, names = list(), list()
    # LR
    models.append(LogisticRegression(solver='lbfgs'))
    names.append('LR')
    # LDA
    models.append(LinearDiscriminantAnalysis())
    names.append('LDA')
    # QDA
```

```

models.append(QuadraticDiscriminantAnalysis())
names.append('QDA')
# GNB
models.append(GaussianNB())
names.append('GNB')
# MNB
models.append(MultinomialNB())
names.append('MNB')
# GPC
models.append(GaussianProcessClassifier())
names.append('GPC')
return models, names

# define the location of the dataset
full_path = 'haberman.csv'
# load the dataset
X, y = load_dataset(full_path)
# define models
models, names = get_models()
results = list()
# evaluate each model
for i in range(len(models)):
    # evaluate the model and store results
    scores = evaluate_model(X, y, models[i])
    results.append(scores)
    # summarize and store
    print('>%s %.3f (%.3f)' % (names[i], mean(scores), std(scores)))
# plot the results
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()

```

Listing 26.22: Example of evaluating probabilistic models for the Haberman’s survival dataset.

Running the example first summarizes the mean and standard deviation of the BSS for each algorithm (larger scores is better).

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, the results suggest that only two of the algorithms are not skillful, showing negative scores, and that perhaps the LR and LDA algorithms are the best performing.

```

>LR 0.060 (0.143)
>LDA 0.064 (0.154)
>QDA 0.027 (0.221)
>GNB 0.012 (0.212)
>MNB -0.211 (0.369)
>GPC -0.142 (0.041)

```

Listing 26.23: Example output from evaluating probabilistic models for the Haberman’s survival dataset.

A box and whisker plot is created summarizing the distribution of results. Interestingly, most if not all algorithms show a spread indicating that they may be unskillful on some of the runs. The distribution between the two top-performing models appears roughly equivalent, so choosing a model based on mean performance might be a good start.

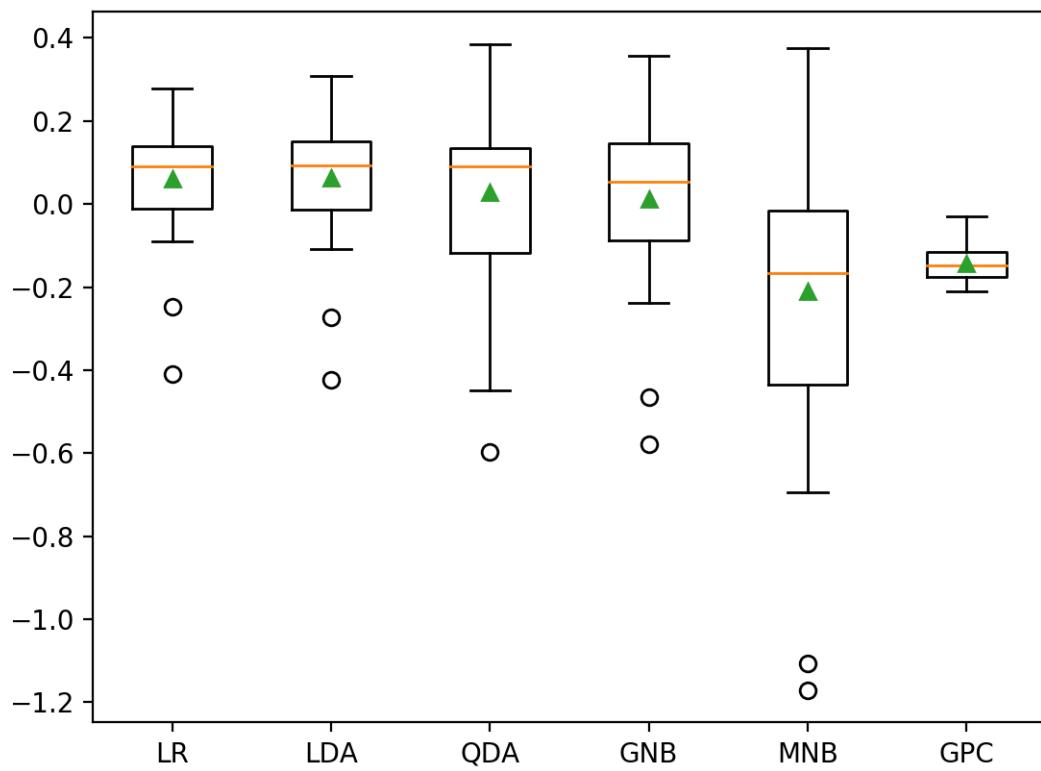


Figure 26.2: Box and Whisker Plot of Probabilistic Models on the Haberman Breast Cancer Survival Dataset.

This is a good start; let's see if we can improve the results with basic data preparation.

26.5.2 Model Evaluation With Scaled Inputs

It can be a good practice to scale data for some algorithms if the variables have different units of measure, as they do in this case. Algorithms like the LR and LDA are sensitive to the distribution of the data and assume a Gaussian distribution for the input variables, which we don't have in all cases.

Nevertheless, we can test the algorithms with standardization, where each variable is shifted to a zero mean and unit standard deviation. We will drop the MNB algorithm as it does not support negative input values. We can achieve this by wrapping each model in a `Pipeline` where the first step is a `StandardScaler`, which will correctly be fit on the training dataset and applied to the test dataset within each k -fold cross-validation evaluation, preventing any data leakage.

```
...
# create a pipeline
pipeline = Pipeline(steps=[('t', StandardScaler()), ('m',models[i])])
# evaluate the model and store results
scores = evaluate_model(X, y, pipeline)
```

Listing 26.24: Example of scaling inputs for defined models.

The complete example of evaluating the five remaining algorithms with standardized input data is listed below.

```
# compare probabilistic models with standardized input on the haberman dataset
from numpy import mean
from numpy import std
from pandas import read_csv
from matplotlib import pyplot
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.metrics import brier_score_loss
from sklearn.metrics import make_scorer
from sklearn.linear_model import LogisticRegression
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

# load the dataset
def load_dataset(full_path):
    # load the dataset as a numpy array
    data = read_csv(full_path, header=None)
    # retrieve numpy array
    data = data.values
    # split into input and output elements
    X, y = data[:, :-1], data[:, -1]
    # label encode the target variable to have the classes 0 and 1
    y = LabelEncoder().fit_transform(y)
    return X, y

# calculate brier skill score (BSS)
def brier_skill_score(y_true, y_prob):
    # calculate reference brier score
    ref_probs = [0.26471 for _ in range(len(y_true))]
    bs_ref = brier_score_loss(y_true, ref_probs)
    # calculate model brier score
    bs_model = brier_score_loss(y_true, y_prob)
    # calculate skill score
    return 1.0 - (bs_model / bs_ref)

# evaluate a model
def evaluate_model(X, y, model):
    # define evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # define the model evaluation metric
    metric = make_scorer(brier_skill_score, needs_proba=True)
    # evaluate model
    scores = cross_val_score(model, X, y, scoring=metric, cv=cv, n_jobs=-1)
    return scores
```

```

# define models to test
def get_models():
    models, names = list(), list()
    # LR
    models.append(LogisticRegression(solver='lbfgs'))
    names.append('LR')
    # LDA
    models.append(LinearDiscriminantAnalysis())
    names.append('LDA')
    # QDA
    models.append(QuadraticDiscriminantAnalysis())
    names.append('QDA')
    # GNB
    models.append(GaussianNB())
    names.append('GNB')
    # GPC
    models.append(GaussianProcessClassifier())
    names.append('GPC')
    return models, names

# define the location of the dataset
full_path = 'haberman.csv'
# load the dataset
X, y = load_dataset(full_path)
# define models
models, names = get_models()
results = list()
# evaluate each model
for i in range(len(models)):
    # create a pipeline
    pipeline = Pipeline(steps=[('t', StandardScaler()), ('m', models[i])])
    # evaluate the model and store results
    scores = evaluate_model(X, y, pipeline)
    results.append(scores)
    # summarize and store
    print('>%s %.3f (%.3f)' % (names[i], mean(scores), std(scores)))
# plot the results
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()

```

Listing 26.25: Example of evaluating probabilistic models with scaled inputs for the Haberman's survival dataset.

Running the example again summarizes the mean and standard deviation of the BSS for each algorithm.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the standardization has not had much of an impact on the algorithms, except the GPC. The performance of the GPC with standardization has shot up and is now the best-performing technique. This highlights the importance of preparing data to meet the expectations of each model.

>LR 0.062 (0.140)

```
>LDA 0.064 (0.154)
>QDA 0.027 (0.221)
>GNB 0.012 (0.212)
>GPC 0.097 (0.133)
```

Listing 26.26: Example output from evaluating probabilistic models with scaled inputs for the Haberman’s survival dataset.

Box and whisker plots for each algorithm’s results are created, showing the difference in mean performance (green triangles) and the similar spread in scores between the three top-performing methods. This suggests all three probabilistic methods are discovering the same general mapping of inputs to probabilities in the dataset.

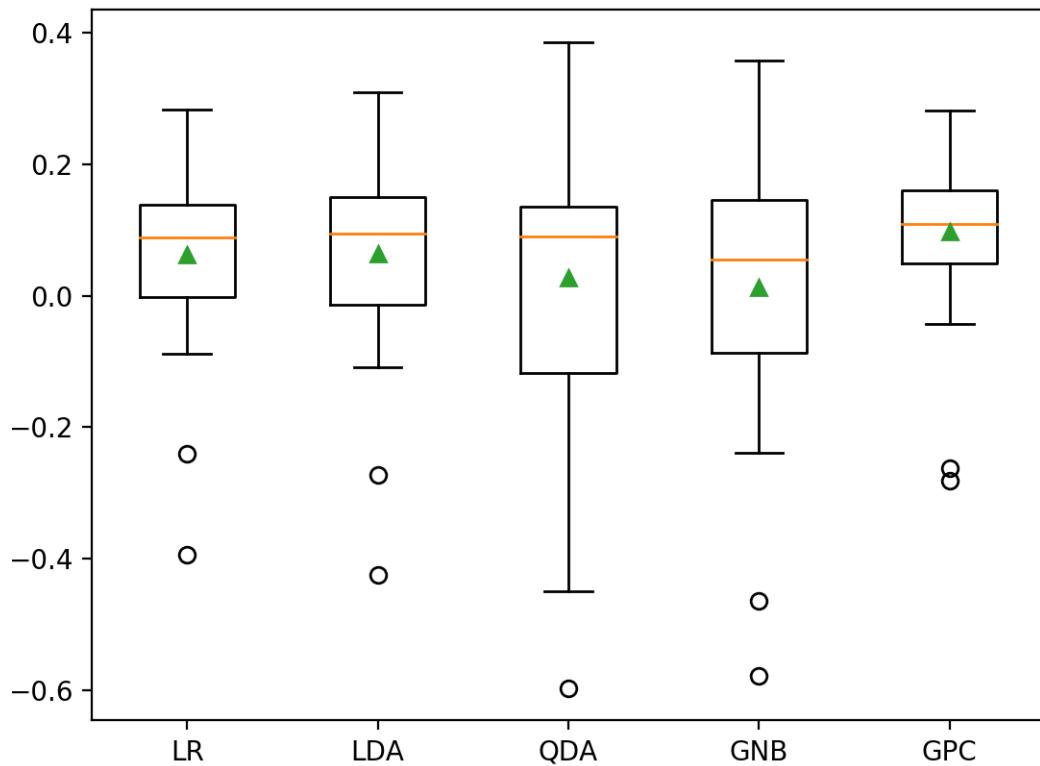


Figure 26.3: Box and Whisker Plot of Probabilistic Models With Data Standardization on the Haberman Breast Cancer Survival Dataset.

There is further data preparation to make the input variables more Gaussian, such as power transforms.

26.5.3 Model Evaluation With Power Transform

Power transforms, such as the Box-Cox and Yeo-Johnson transforms, are designed to change the distribution to be more Gaussian. This will help with the *age* input variable in our dataset

and may help with the *nodes* variable and un-bunch the distribution slightly. We can use the `PowerTransformer` scikit-learn class to perform the Yeo-Johnson transform and automatically determine the best parameters to apply based on the dataset, e.g. how to best make each variable more Gaussian. Importantly, this transformer will also standardize the dataset as part of the transform, ensuring we keep the gains seen in the previous section.

The power transform may make use of a `log()` function, which does not work on zero values. We have zero values in our dataset, therefore we will scale the dataset prior to the power transform using a `MinMaxScaler`. Again, we can use this transform in a `Pipeline` to ensure it is fit on the training dataset and applied to the train and test datasets correctly, without data leakage.

```
...
# create a pipeline
steps = [('t1', MinMaxScaler()), ('t2', PowerTransformer()), ('m', models[i])]
pipeline = Pipeline(steps=steps)
# evaluate the model and store results
scores = evaluate_model(X, y, pipeline)
```

Listing 26.27: Example of applying a power transform to defined models.

We will focus on the three top-performing methods, in this case, LR, LDA, and GPC. The complete example is listed below.

```
# compare probabilistic models with power transforms on the haberman dataset
from numpy import mean
from numpy import std
from pandas import read_csv
from matplotlib import pyplot
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.metrics import brier_score_loss
from sklearn.metrics import make_scorer
from sklearn.linear_model import LogisticRegression
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PowerTransformer
from sklearn.preprocessing import MinMaxScaler

# load the dataset
def load_dataset(full_path):
    # load the dataset as a numpy array
    data = read_csv(full_path, header=None)
    # retrieve numpy array
    data = data.values
    # split into input and output elements
    X, y = data[:, :-1], data[:, -1]
    # label encode the target variable to have the classes 0 and 1
    y = LabelEncoder().fit_transform(y)
    return X, y

# calculate brier skill score (BSS)
def brier_skill_score(y_true, y_prob):
    # calculate reference brier score
```

```

ref_probs = [0.26471 for _ in range(len(y_true))]
bs_ref = brier_score_loss(y_true, ref_probs)
# calculate model brier score
bs_model = brier_score_loss(y_true, y_prob)
# calculate skill score
return 1.0 - (bs_model / bs_ref)

# evaluate a model
def evaluate_model(X, y, model):
    # define evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # define the model evaluation metric
    metric = make_scorer(brier_skill_score, needs_proba=True)
    # evaluate model
    scores = cross_val_score(model, X, y, scoring=metric, cv=cv, n_jobs=-1)
    return scores

# define models to test
def get_models():
    models, names = list(), list()
    # LR
    models.append(LogisticRegression(solver='lbfgs'))
    names.append('LR')
    # LDA
    models.append(LinearDiscriminantAnalysis())
    names.append('LDA')
    # GPC
    models.append(GaussianProcessClassifier())
    names.append('GPC')
    return models, names

# define the location of the dataset
full_path = 'haberman.csv'
# load the dataset
X, y = load_dataset(full_path)
# define models
models, names = get_models()
results = list()
# evaluate each model
for i in range(len(models)):
    # create a pipeline
    steps = [('t1', MinMaxScaler()), ('t2', PowerTransformer()), ('m', models[i])]
    pipeline = Pipeline(steps=steps)
    # evaluate the model and store results
    scores = evaluate_model(X, y, pipeline)
    results.append(scores)
    # summarize and store
    print('>%s %.3f (%.3f)' % (names[i], mean(scores), std(scores)))
# plot the results
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()

```

Listing 26.28: Example of evaluating probabilistic models with power transformed inputs for the Haberman's survival dataset.

Running the example again summarizes the mean and standard deviation of the BSS for

each algorithm.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see a further lift in model skill for the three models that were evaluated. We can see that the LR appears to have out-performed the other two methods.

```
>LR 0.110 (0.142)
>LDA 0.107 (0.164)
>GPC 0.100 (0.130)
```

Listing 26.29: Example output from evaluating probabilistic models with power transformed inputs for the Haberman's survival dataset.

Box and whisker plots are created for the results from each algorithm, suggesting perhaps a smaller and more focused spread for LR compared to the LDA, which was the second-best performing method. All methods still show skill on average, however the distribution of scores show runs that drop below 0.0 (no skill) in some cases.

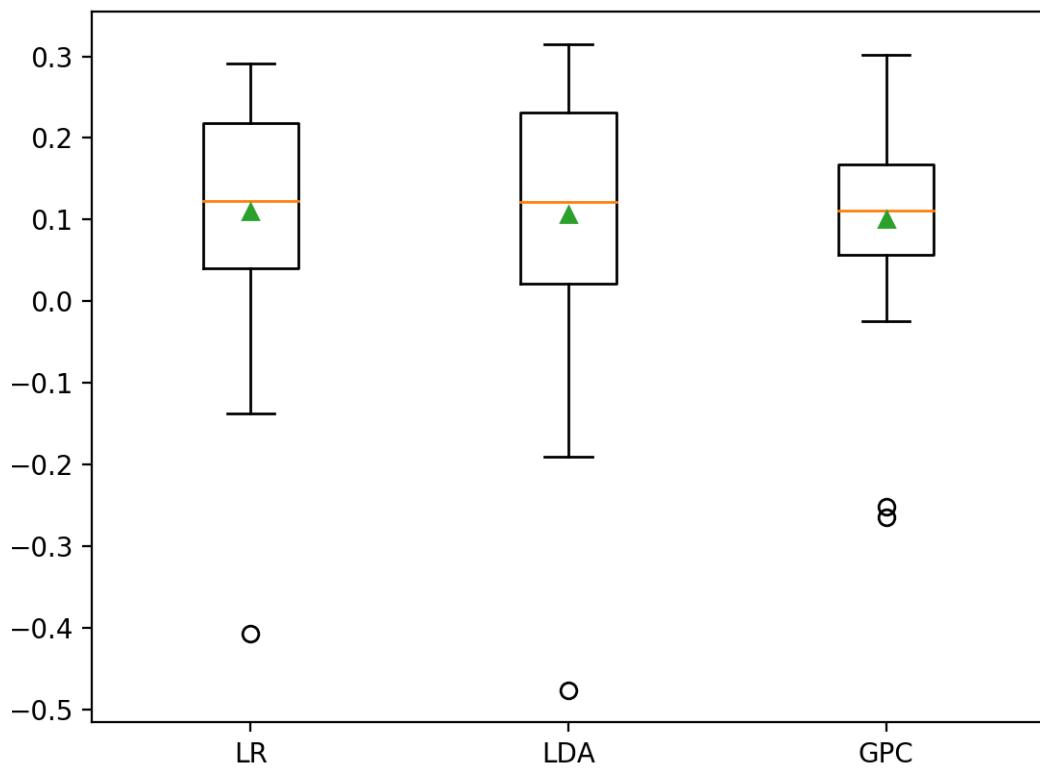


Figure 26.4: Box and Whisker Plot of Probabilistic Models With Power Transforms on the Haberman Breast Cancer Survival Dataset.

26.6 Make Prediction on New Data

We will select the Logistic Regression model with a power transform on the input data as our final model. We can define and fit this model on the entire training dataset.

```
...
# fit the model
steps = [('t1', MinMaxScaler()), ('t2',
    PowerTransformer()), ('m', LogisticRegression(solver='lbfgs'))]
model = Pipeline(steps=steps)
model.fit(X, y)
```

Listing 26.30: Example of defining the pipeline for the final model.

Once fit, we can use it to make predictions for new data by calling the `predict_proba()` function. This will return two probabilities for each prediction, the first for survival and the second for non-survival, e.g. its complement. For example:

```
...
row = [31,59,2]
yhat = model.predict_proba([row])
# get percentage of survival
p_survive = yhat[0, 0] * 100
```

Listing 26.31: Example of making a prediction with the final model.

To demonstrate this, we can use the fit model to make some predictions of probability for a few cases where we know there is survival and a few where we know there is not. The complete example is listed below.

```
# fit a model and make predictions for the haberman dataset
from pandas import read_csv
from sklearn.preprocessing import LabelEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PowerTransformer
from sklearn.preprocessing import MinMaxScaler

# load the dataset
def load_dataset(full_path):
    # load the dataset as a numpy array
    data = read_csv(full_path, header=None)
    # retrieve numpy array
    data = data.values
    # split into input and output elements
    X, y = data[:, :-1], data[:, -1]
    # label encode the target variable to have the classes 0 and 1
    y = LabelEncoder().fit_transform(y)
    return X, y

# define the location of the dataset
full_path = 'haberman.csv'
# load the dataset
X, y = load_dataset(full_path)
# fit the model
steps = [('t1', MinMaxScaler()), ('t2',
    PowerTransformer()), ('m', LogisticRegression(solver='lbfgs'))]
```

```

model = Pipeline(steps=steps)
model.fit(X, y)
# some survival cases
print('Survival Cases:')
data = [[31,59,2], [31,65,4], [34,60,1]]
for row in data:
    # make prediction
    yhat = model.predict_proba([row])
    # get percentage of survival
    p_survive = yhat[0, 0] * 100
    # summarize
    print('>data=%s, Survival=%.3f%%' % (row, p_survive))
# some non-survival cases
print('Non-Survival Cases:')
data = [[44,64,6], [34,66,9], [38,69,21]]
for row in data:
    # make prediction
    yhat = model.predict_proba([row])
    # get percentage of survival
    p_survive = yhat[0, 0] * 100
    # summarize
    print('>data=%s, Survival=%.3f%%' % (row, p_survive))

```

Listing 26.32: Example fitting the final model and making predictions for the Haberman’s survival dataset.

Running the example first fits the model on the entire dataset. Then the fit model is used to predict the probability of survival for cases where we know the patient survived, chosen from the dataset file. We can see that for the chosen survival cases, the probability of survival was high, between 77 percent and 86 percent. Then some cases of non-survival are used as input to the model and the probability of survival is predicted. As we might have hoped, the probability of survival is modest, hovering around 53 percent to 63 percent.

```

Survival Cases:
>data=[31, 59, 2], Survival=83.597%
>data=[31, 65, 4], Survival=77.264%
>data=[34, 60, 1], Survival=86.776%
Non-Survival Cases:
>data=[44, 64, 6], Survival=63.092%
>data=[34, 66, 9], Survival=63.452%
>data=[38, 69, 21], Survival=53.389%

```

Listing 26.33: Example output from fitting the final model and making predictions for the Haberman’s survival dataset.

26.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

26.7.1 APIs

- `pandas.read_csv` API.
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html.

[html](#)

- `pandas.DataFrame.describe` API.
<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.describe.html>
- `pandas.DataFrame.hist` API.
<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.hist.html>
- `sklearn.preprocessing.LabelEncoder` API.
<https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html>
- `sklearn.preprocessing.PowerTransformer` API.
<https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PowerTransformer.html>

26.7.2 Dataset

- Haberman's Survival Data Set, UCI Machine Learning Repository.
<https://archive.ics.uci.edu/ml/datasets/Haberman's+Survival>
- Haberman's Survival Data Set CSV File.
<https://raw.githubusercontent.com/jbrownlee/Datasets/master/haberman.csv>
- Haberman's Survival Data Set Names File.
<https://raw.githubusercontent.com/jbrownlee/Datasets/master/haberman.names>

26.8 Summary

In this tutorial, you discovered how to develop a model to predict the probability of patient survival on an imbalanced dataset. Specifically, you learned:

- How to load and explore the dataset and generate ideas for data preparation and model selection.
- How to evaluate a suite of probabilistic models and improve their performance with appropriate data preparation.
- How to fit a final model and use it to predict probabilities for specific cases.

26.8.1 Next

In the next tutorial, you will discover how to systematically work through the oil spill imbalanced classification dataset.

Chapter 27

Project: Oil Spill Classification

Many imbalanced classification tasks require a skillful model that predicts a crisp class label, where both classes are equally important. An example of an imbalanced classification problem where a class label is required and both classes are equally important is the detection of oil spills or slicks in satellite images. The detection of a spill requires mobilizing an expensive response, and missing an event is equally expensive, causing damage to the environment.

One way to evaluate imbalanced classification models that predict crisp labels is to calculate the separate accuracy on the positive class and the negative class, referred to as sensitivity and specificity. These two measures can then be averaged using the geometric mean, referred to as the G-mean, that is insensitive to the skewed class distribution and correctly reports on the skill of the model on both classes. In this tutorial, you will discover how to develop a model to predict the presence of an oil spill in satellite images and evaluate it using the G-mean metric. After completing this tutorial, you will know:

- How to load and explore the dataset and generate ideas for data preparation and model selection.
- How to evaluate a suite of probabilistic models and improve their performance with appropriate data preparation.
- How to fit a final model and use it to predict class labels for specific cases.

Let's get started.

Note: This chapter makes use of the imbalanced-learn library. See Appendix B for installation instructions, if needed.

27.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Oil Spill Dataset
2. Explore the Dataset
3. Model Test and Baseline Result

4. Evaluate Models
5. Make Prediction on New Data

27.2 Oil Spill Dataset

In this project, we will use a standard imbalanced machine learning dataset referred to as the *oil spill* dataset, *oil slicks* dataset or simply *oil*. The dataset was introduced in the 1998 paper by Miroslav Kubat, et al. titled *Machine Learning for the Detection of Oil Spills in Satellite Radar Images*. The dataset is often credited to Robert Holte, a co-author of the paper. The dataset was developed by starting with satellite images of the ocean, some of which contain an oil spill and some that do not. Images were split into sections and processed using computer vision algorithms to provide a vector of features to describe the contents of the image section or patch.

The input to [the system] is a raw pixel image from a radar satellite. Image processing techniques are used [...] The output of the image processing is a fixed-length feature vector for each suspicious region. During normal operation these feature vectors are fed into a classifier to decide which images and which regions within an image to present for human inspection.

— *Machine Learning for the Detection of Oil Spills in Satellite Radar Images*, 1998.

In the task, a model is given a vector that describes the contents of a patch of a satellite image, then predicts whether the patch contains an oil spill or not, e.g. from the illegal or accidental dumping of oil in the ocean. There are 937 cases. Each case is comprised of 48 numerical computer vision derived features, a patch number, and a class label.

A total of nine satellite images were processed into patches. Cases in the dataset are ordered by image and the first column of the dataset represents the patch number for the image. This was provided for the purposes of estimating model performance per-image. In this case, we are not interested in the image or patch number and this first column can be removed. The normal case is no oil spill assigned the class label of 0, whereas an oil spill is indicated by a class label of 1. There are 896 cases for no oil spill and 41 cases of an oil spill.

The second critical feature of the oil spill domain can be called an imbalanced training set: there are very many more negative examples lookalikes than positive examples oil slicks. Against the 41 positive examples we have 896 negative examples the majority class thus comprises almost 96% of the data.

— *Machine Learning for the Detection of Oil Spills in Satellite Radar Images*, 1998.

We do not have access to the program used to prepare computer vision features from the satellite images, therefore we are restricted to work with the extracted features that were collected and made available. Next, let's take a closer look at the data.

27.3 Explore the Dataset

First, download the dataset and save it in your current working directory with the name `oil-spill.csv`.

- Download Oil Spill Dataset (`oil-spill.csv`). ¹

Review the contents of the file. We can see that the first column contains integers for the patch number. We can also see that the computer vision derived features are real-valued with differing scales such as thousands in the second column and fractions in other columns. All input variables are numeric. Firstly, we can load the CSV dataset and confirm the number of rows and columns. The dataset can be loaded as a `DataFrame` using the `read_csv()` Pandas function, specifying the location and the fact that there is no header line.

```
...
# define the dataset location
filename = 'oil-spill.csv'
# load the csv file as a data frame
dataframe = read_csv(filename, header=None)
```

Listing 27.1: Example of loading the oil dataset.

Once loaded, we can summarize the number of rows and columns by printing the shape of the `DataFrame`.

```
...
# summarize the shape of the dataset
print(dataframe.shape)
```

Listing 27.2: Example of summarizing the shape of the loaded dataset.

We can also summarize the number of examples in each class using the `Counter` object.

```
...
# summarize the class distribution
target = dataframe.values[:, -1]
counter = Counter(target)
for k, v in counter.items():
    per = v / len(target) * 100
    print('Class=%d, Count=%d, Percentage=%.3f%%' % (k, v, per))
```

Listing 27.3: Example of summarizing the class distribution for the loaded dataset.

Tying this together, the complete example of loading and summarizing the dataset is listed below.

```
# load and summarize the dataset
from pandas import read_csv
from collections import Counter
# define the dataset location
filename = 'oil-spill.csv'
# load the csv file as a data frame
dataframe = read_csv(filename, header=None)
# summarize the shape of the dataset
```

¹<https://raw.githubusercontent.com/jbrownlee/Datasets/master/oil-spill.csv>

```

print(dataframe.shape)
# summarize the class distribution
target = dataframe.values[:, -1]
counter = Counter(target)
for k, v in counter.items():
    per = v / len(target) * 100
    print('Class=%d, Count=%d, Percentage=%.3f%%' % (k, v, per))

```

Listing 27.4: Load and summarize the oil dataset.

Running the example first loads the dataset and confirms the number of rows and columns. The class distribution is then summarized, confirming the number of oil spills and non-spills and the percentage of cases in the minority and majority classes.

```
(937, 50)
Class=1, Count=41, Percentage=4.376%
Class=0, Count=896, Percentage=95.624%
```

Listing 27.5: Example output from loading and summarizing the oil dataset.

We can also take a look at the distribution of each variable by creating a histogram for each. With 50 variables, it is a lot of plots, but we might spot some interesting patterns. Also, with so many plots, we must turn off the axis labels and plot titles to reduce the clutter. The complete example is listed below.

```

# create histograms of each variable
from pandas import read_csv
from matplotlib import pyplot
# define the dataset location
filename = 'oil-spill.csv'
# load the csv file as a data frame
dataframe = read_csv(filename, header=None)
# create a histogram plot of each variable
ax = dataframe.hist()
# disable axis labels
for axis in ax.flatten():
    axis.set_title('')
    axis.set_xticklabels([])
    axis.set_yticklabels([])
pyplot.show()

```

Listing 27.6: Plot histograms of the oil dataset.

Running the example creates the figure with one histogram subplot for each of the 50 variables in the dataset. We can see many different distributions, some with Gaussian-like distributions, others with seemingly exponential or discrete distributions. Depending on the choice of modeling algorithms, we would expect scaling the distributions to the same range to be useful, and perhaps the use of some power transforms.

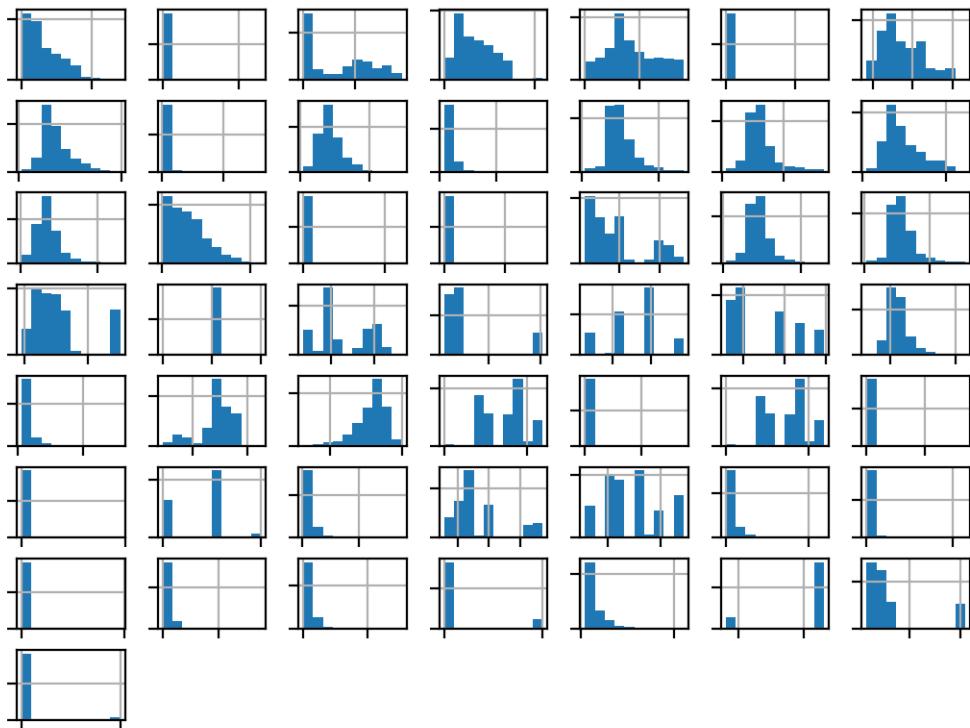


Figure 27.1: Histogram of Each Variable in the Oil Spill Dataset.

Now that we have reviewed the dataset, let's look at developing a test harness for evaluating candidate models.

27.4 Model Test and Baseline Result

We will evaluate candidate models using repeated stratified k -fold cross-validation. The k -fold cross-validation procedure provides a good general estimate of model performance that is not too optimistically biased, at least compared to a single train-test split. We will use $k=10$, meaning each fold will contain about $\frac{937}{10}$ or about 94 examples. Stratified means that each fold will contain the same mixture of examples by class, that is about 96% to 4% non-spill and spill. Repeated means that the evaluation process will be performed multiple times to help avoid fluke results and better capture the variance of the chosen model. We will use three repeats. This means a single model will be fit and evaluated 10×3 (30) times and the mean and standard deviation of these runs will be reported.

This can be achieved using the `RepeatedStratifiedKFold` scikit-learn class. We are predicting class labels of whether a satellite image patch contains a spill or not. There are many measures we could use, although the authors of the paper chose to report the sensitivity, specificity, and the geometric mean of the two scores, called the G-mean.

To this end, we have mainly used the geometric mean (g-mean) [...] This measure has the distinctive property of being independent of the distribution of examples between classes, and is thus robust in circumstances where this distribution might change with time or be different in the training and testing sets.

— *Machine Learning for the Detection of Oil Spills in Satellite Radar Images*, 1998.

Recall that the sensitivity is a measure of the accuracy for the positive class and specificity is a measure of the accuracy of the negative class. The G-mean seeks a balance of these scores, the geometric mean, where poor performance for one or the other results in a low G-mean score (for more on the G-mean, see Chapter 4). We can calculate the G-mean for a set of predictions made by a model using the `geometric_mean_score()` function provided by the imbalanced-learn library. First, we can define a function to load the dataset and split the columns into input and output variables. We will also drop column 22 because the column contains a single value, and the first column that defines the image patch number. The `load_dataset()` function below implements this.

```
# load the dataset
def load_dataset(full_path):
    # load the dataset as a numpy array
    data = read_csv(full_path, header=None)
    # drop unused columns
    data.drop(22, axis=1, inplace=True)
    data.drop(0, axis=1, inplace=True)
    # retrieve numpy array
    data = data.values
    # split into input and output elements
    X, y = data[:, :-1], data[:, -1]
    # label encode the target variable to have the classes 0 and 1
    y = LabelEncoder().fit_transform(y)
    return X, y
```

Listing 27.7: Example a function for loading and preparing the dataset.

We can then define a function that will evaluate a given model on the dataset and return a list of G-mean scores for each fold and repeat. The `evaluate_model()` function below implements this, taking the dataset and model as arguments and returning the list of scores.

```
# evaluate a model
def evaluate_model(X, y, model):
    # define evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # define the model evaluation metric
    metric = make_scorer(geometric_mean_score)
    # evaluate model
    scores = cross_val_score(model, X, y, scoring=metric, cv=cv, n_jobs=-1)
    return scores
```

Listing 27.8: Example of a function for evaluating a model.

Finally, we can evaluate a baseline model on the dataset using this test harness. A model that predicts the majority class label (0) or the minority class label (1) for all cases will result in a G-mean of zero. As such, a good default strategy would be to randomly predict one class label or another with a 50% probability and aim for a G-mean of about 0.5. This can be achieved using

the `DummyClassifier` class from the scikit-learn library and setting the `strategy` argument to ‘uniform’.

```
...
# define the reference model
model = DummyClassifier(strategy='uniform')
```

Listing 27.9: Example of defining the baseline model.

Once the model is evaluated, we can report the mean and standard deviation of the G-mean scores directly.

```
...
# evaluate the model
result_m, result_s = evaluate_model(X, y, model)
# summarize performance
print('Mean G-mean: %.3f (%.3f)' % (result_m, result_s))
```

Listing 27.10: Example of evaluating the model and summarizing the performance.

Tying this together, the complete example of loading the dataset, evaluating a baseline model and reporting the performance is listed below.

```
# test harness and baseline model evaluation
from collections import Counter
from numpy import mean
from numpy import std
from pandas import read_csv
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from imblearn.metrics import geometric_mean_score
from sklearn.metrics import make_scorer
from sklearn.dummy import DummyClassifier

# load the dataset
def load_dataset(full_path):
    # load the dataset as a numpy array
    data = read_csv(full_path, header=None)
    # drop unused columns
    data.drop(22, axis=1, inplace=True)
    data.drop(0, axis=1, inplace=True)
    # retrieve numpy array
    data = data.values
    # split into input and output elements
    X, y = data[:, :-1], data[:, -1]
    # label encode the target variable to have the classes 0 and 1
    y = LabelEncoder().fit_transform(y)
    return X, y

# evaluate a model
def evaluate_model(X, y, model):
    # define evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # define the model evaluation the metric
    metric = make_scorer(geometric_mean_score)
    # evaluate model
```

```

scores = cross_val_score(model, X, y, scoring=metric, cv=cv, n_jobs=-1)
return scores

# define the location of the dataset
full_path = 'oil-spill.csv'
# load the dataset
X, y = load_dataset(full_path)
# summarize the loaded dataset
print(X.shape, y.shape, Counter(y))
# define the reference model
model = DummyClassifier(strategy='uniform')
# evaluate the model
scores = evaluate_model(X, y, model)
# summarize performance
print('Mean G-Mean: %.3f (%.3f)' % (mean(scores), std(scores)))

```

Listing 27.11: Calculate the baseline performance for the oil dataset.

Running the example first loads and summarizes the dataset. We can see that we have the correct number of rows loaded, and that we have 47 computer vision derived input variables, with the constant value column (index 22) and the patch number column (index 0) removed. Importantly, we can see that the class labels have the correct mapping to integers with 0 for the majority class and 1 for the minority class, customary for imbalanced binary classification dataset. Next, the average of the G-mean scores is reported.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the baseline algorithm achieves a G-mean of about 0.47, close to the theoretical maximum of 0.5. This score provides a lower limit on model skill; any model that achieves an average G-mean above about 0.47 (or really above 0.5) has skill, whereas models that achieve a score below this value do not have skill on this dataset.

```
(937, 47) (937,) Counter({0: 896, 1: 41})
Mean G-mean: 0.478 (0.143)
```

Listing 27.12: Example output from calculating the baseline performance for the oil dataset.

It is interesting to note that a good G-mean reported in the paper was about 0.811, although the model evaluation procedure was different. This provides a rough target for *good* performance on this dataset. Now that we have a test harness and a baseline in performance, we can begin to evaluate some models on this dataset.

27.5 Evaluate Models

In this section, we will evaluate a suite of different techniques on the dataset using the test harness developed in the previous section. The goal is to both demonstrate how to work through the problem systematically and to demonstrate the capability of some techniques designed for imbalanced classification problems. The reported performance is good, but not highly optimized (e.g. hyperparameters are not tuned).

27.5.1 Evaluate Probabilistic Models

Let's start by evaluating some probabilistic models on the dataset. Probabilistic models are those models that are fit on the data under a probabilistic framework and often perform well in general for imbalanced classification datasets. We will evaluate the following probabilistic models with default hyperparameters in the dataset:

- Logistic Regression (LR)
- Linear Discriminant Analysis (LDA)
- Gaussian Naive Bayes (NB)

Both LR and LDA are sensitive to the scale of the input variables, and often expect and/or perform better if input variables with different scales are normalized or standardized as a pre-processing step. In this case, we will standardize the dataset prior to fitting each model. This will be achieved using a `Pipeline` and the `StandardScaler` class. The use of a `Pipeline` ensures that the `StandardScaler` is fit on the training dataset and applied to the train and test sets within each k -fold cross-validation evaluation, avoiding any data leakage that might result in an optimistic result. We can define a function to create the models to evaluate on our test harness as follows:

```
# define models to test
def get_models():
    models, names = list(), list()
    # LR
    steps = [('t', StandardScaler()), ('m', LogisticRegression(solver='liblinear'))]
    models.append(Pipeline(steps=steps))
    names.append('LR')
    # LDA
    steps = [('t', StandardScaler()), ('m', LinearDiscriminantAnalysis())]
    models.append(Pipeline(steps=steps))
    names.append('LDA')
    # NB
    models.append(GaussianNB())
    names.append('NB')
    return models, names
```

Listing 27.13: Example of a function defining the models to evaluate.

Once defined, we can enumerate the list and evaluate each in turn. The mean and standard deviation of G-mean scores can be printed during evaluation and the sample of scores can be stored. Algorithms can be compared directly based on their mean G-mean score.

```
...
# evaluate each model
for i in range(len(models)):
    # evaluate the model and store results
    scores = evaluate_model(X, y, models[i])
    results.append(scores)
    # summarize and store
    print('>%s %.3f (%.3f)' % (names[i], mean(scores), std(scores)))
```

Listing 27.14: Example of evaluating the defined models and summarizing performance.

At the end of the run, we can use the scores to create a box and whisker plot for each algorithm. Creating the plots side by side allows the distributions to be compared both with regard to the mean score, but also the middle 50 percent of the distribution between the 25th and 75th percentiles.

```
...
# plot the results
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 27.15: Example of summarizing performance using box and whisker plots.

Tying this together, the complete example comparing three probabilistic models on the oil spill dataset using the test harness is listed below.

```
# compare probabilistic model on the oil spill dataset
from numpy import mean
from numpy import std
from pandas import read_csv
from matplotlib import pyplot
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.metrics import make_scorer
from sklearn.linear_model import LogisticRegression
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from imblearn.metrics import geometric_mean_score
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

# load the dataset
def load_dataset(full_path):
    # load the dataset as a numpy array
    data = read_csv(full_path, header=None)
    # drop unused columns
    data.drop(22, axis=1, inplace=True)
    data.drop(0, axis=1, inplace=True)
    # retrieve numpy array
    data = data.values
    # split into input and output elements
    X, y = data[:, :-1], data[:, -1]
    # label encode the target variable to have the classes 0 and 1
    y = LabelEncoder().fit_transform(y)
    return X, y

# evaluate a model
def evaluate_model(X, y, model):
    # define evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # define the model evaluation the metric
    metric = make_scorer(geometric_mean_score)
    # evaluate model
    scores = cross_val_score(model, X, y, scoring=metric, cv=cv, n_jobs=-1)
    return scores
```

```

# define models to test
def get_models():
    models, names = list(), list()
    # LR
    steps = [('t', StandardScaler()), ('m', LogisticRegression(solver='liblinear'))]
    models.append(Pipeline(steps=steps))
    names.append('LR')
    # LDA
    steps = [('t', StandardScaler()), ('m', LinearDiscriminantAnalysis())]
    models.append(Pipeline(steps=steps))
    names.append('LDA')
    # NB
    models.append(GaussianNB())
    names.append('NB')
    return models, names

# define the location of the dataset
full_path = 'oil-spill.csv'
# load the dataset
X, y = load_dataset(full_path)
# define models
models, names = get_models()
# evaluate each model
results = list()
for i in range(len(models)):
    # evaluate the model and store results
    scores = evaluate_model(X, y, models[i])
    results.append(scores)
    # summarize and store
    print('>%s %.3f (%.3f)' % (names[i], mean(scores), std(scores)))
# plot the results
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()

```

Listing 27.16: Calculate the performance of probabilistic models on the oil dataset.

Running the example evaluates each of the probabilistic models on the dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

You may see some warnings from the LDA algorithm such as: *Variables are collinear*. These can be safely ignored for now, but suggests that the algorithm could benefit from feature selection to remove some of the variables. In this case, we can see that each algorithm has skill, achieving a mean G-mean above 0.5. The results suggest that an LDA might be the best performing of the models tested.

```

>LR 0.621 (0.261)
>LDA 0.741 (0.220)
>NB 0.721 (0.197)

```

Listing 27.17: Example output from calculating the performance of probabilistic models on the oil dataset.

The distribution of the G-mean scores is summarized using a figure with a box and whisker plot for each algorithm. We can see that the distribution for both LDA and NB is compact and

skillful and that the LR may have a few results during the run where the method performed poorly, pushing the distribution down. This highlights that it is not just the mean performance, but also the consistency of the model that should be considered when selecting a model.

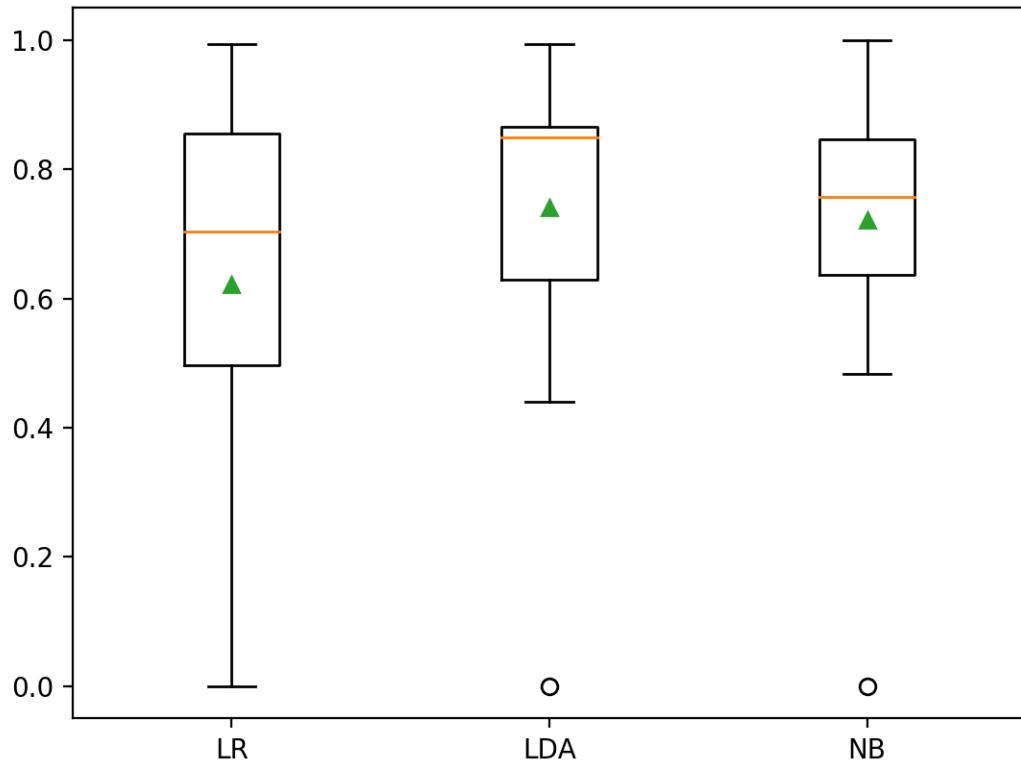


Figure 27.2: Box and Whisker Plot of Probabilistic Models on the Imbalanced Oil Spill Dataset.

We're off to a good start, but we can do better.

27.5.2 Evaluate Balanced Logistic Regression

The logistic regression algorithm supports a modification that adjusts the importance of classification errors to be inversely proportional to the class weighting. This allows the model to better learn the class boundary in favor of the minority class, which might help overall G-mean performance. We can achieve this by setting the `class_weight` argument of the `LogisticRegression` to ‘balanced’.

```
...
LogisticRegression(solver='liblinear', class_weight='balanced')
```

Listing 27.18: Example of defining a balanced logistic regression model.

As mentioned, logistic regression is sensitive to the scale of input variables and can perform better with normalized or standardized inputs; as such it is a good idea to test both for a given dataset. Additionally, a power distribution can be used to spread out the distribution of each

input variable and make those variables with a Gaussian-like distribution more Gaussian. This can benefit models like Logistic Regression that make assumptions about the distribution of input variables.

The power transform will use the Yeo-Johnson method that supports positive and negative inputs, but we will also normalize data prior to the transform. Also, the `PowerTransformer` class used for the transform will also standardize each variable after the transform. We will compare a `LogisticRegression` with a balanced class weighting to the same algorithm with three different data preparation schemes, specifically normalization, standardization, and a power transform.

```
# define models to test
def get_models():
    models, names = list(), list()
    # LR Balanced
    models.append(LogisticRegression(solver='liblinear', class_weight='balanced'))
    names.append('Balanced')
    # LR Balanced + Normalization
    steps = [('t', MinMaxScaler()), ('m', LogisticRegression(solver='liblinear',
        class_weight='balanced'))]
    models.append(Pipeline(steps=steps))
    names.append('Balanced-Norm')
    # LR Balanced + Standardization
    steps = [('t', StandardScaler()), ('m', LogisticRegression(solver='liblinear',
        class_weight='balanced'))]
    models.append(Pipeline(steps=steps))
    names.append('Balanced-Std')
    # LR Balanced + Power
    steps = [('t1', MinMaxScaler()), ('t2', PowerTransformer()), ('m',
        LogisticRegression(solver='liblinear', class_weight='balanced'))]
    models.append(Pipeline(steps=steps))
    names.append('Balanced-Power')
    return models, names
```

Listing 27.19: Example of defining model pipelines to evaluate.

Tying this together, the comparison of balanced logistic regression with different data preparation schemes is listed below.

```
# compare balanced logistic regression on the oil spill dataset
from numpy import mean
from numpy import std
from pandas import read_csv
from matplotlib import pyplot
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.metrics import make_scorer
from sklearn.linear_model import LogisticRegression
from imblearn.metrics import geometric_mean_score
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import PowerTransformer

# load the dataset
```

```
def load_dataset(full_path):
    # load the dataset as a numpy array
    data = read_csv(full_path, header=None)
    # drop unused columns
    data.drop(22, axis=1, inplace=True)
    data.drop(0, axis=1, inplace=True)
    # retrieve numpy array
    data = data.values
    # split into input and output elements
    X, y = data[:, :-1], data[:, -1]
    # label encode the target variable to have the classes 0 and 1
    y = LabelEncoder().fit_transform(y)
    return X, y

# evaluate a model
def evaluate_model(X, y, model):
    # define evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # define the model evaluation the metric
    metric = make_scorer(geometric_mean_score)
    # evaluate model
    scores = cross_val_score(model, X, y, scoring=metric, cv=cv, n_jobs=-1)
    return scores

# define models to test
def get_models():
    models, names = list(), list()
    # LR Balanced
    models.append(LogisticRegression(solver='liblinear', class_weight='balanced'))
    names.append('Balanced')
    # LR Balanced + Normalization
    steps = [('t',MinMaxScaler()), ('m', LogisticRegression(solver='liblinear',
        class_weight='balanced'))]
    models.append(Pipeline(steps=steps))
    names.append('Balanced-Norm')
    # LR Balanced + Standardization
    steps = [('t',StandardScaler()), ('m', LogisticRegression(solver='liblinear',
        class_weight='balanced'))]
    models.append(Pipeline(steps=steps))
    names.append('Balanced-Std')
    # LR Balanced + Power
    steps = [('t1',MinMaxScaler()), ('t2',PowerTransformer()), ('m',
        LogisticRegression(solver='liblinear', class_weight='balanced'))]
    models.append(Pipeline(steps=steps))
    names.append('Balanced-Power')
    return models, names

# define the location of the dataset
full_path = 'oil-spill.csv'
# load the dataset
X, y = load_dataset(full_path)
# define models
models, names = get_models()
# evaluate each model
results = list()
for i in range(len(models)):
```

```

# evaluate the model and store results
scores = evaluate_model(X, y, models[i])
results.append(scores)
# summarize and store
print('>%s %.3f (%.3f)' % (names[i], mean(scores), std(scores)))
# plot the results
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()

```

Listing 27.20: Calculate the performance of balanced probabilistic models on the oil dataset.

Running the example evaluates each version of the balanced logistic regression model on the dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

You may see some warnings from the first balanced LR model, such as *Liblinear failed to converge*. These warnings can be safely ignored for now but suggest that the algorithm could benefit from feature selection to remove some of the variables.

In this case, we can see that the balanced version of logistic regression performs much better than all of the probabilistic models evaluated in the previous section. The results suggest that perhaps the use of balanced LR with data normalization for pre-processing performs the best on this dataset with a mean G-mean score of about 0.852. This is in the range or better than the results reported in the 1998 paper.

```

>Balanced 0.846 (0.142)
>Balanced-Norm 0.852 (0.119)
>Balanced-Std 0.843 (0.124)
>Balanced-Power 0.847 (0.130)

```

Listing 27.21: Example output from calculating the performance of balanced probabilistic models on the oil dataset.

A figure is created with box and whisker plots for each algorithm, allowing the distribution of results to be compared. We can see that the distribution for the balanced LR is tighter in general than the non-balanced version in the previous section. We can also see that the median result (orange line) for the normalized version is higher than the mean, above 0.9, which is impressive. A mean different from the median suggests a skewed distribution for the results, pulling the mean down with a few bad outcomes.

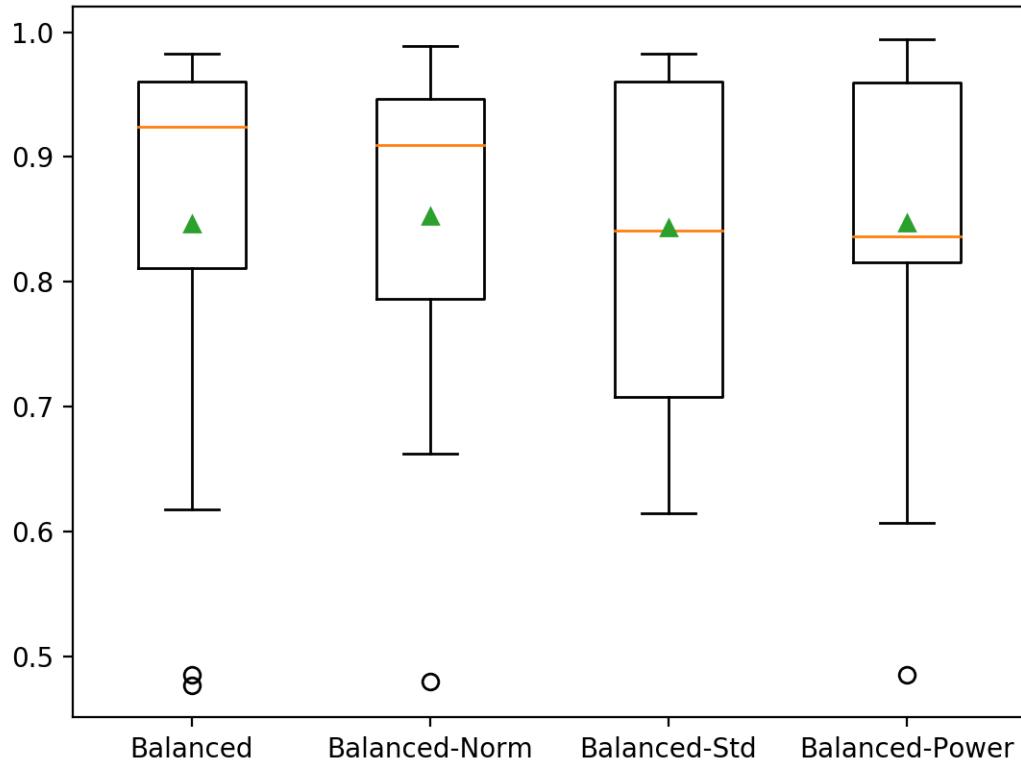


Figure 27.3: Box and Whisker Plot of Balanced Logistic Regression Models on the Imbalanced Oil Spill Dataset.

We now have excellent results with little work; let's see if we can take it one step further.

27.5.3 Evaluate Data Sampling With Probabilistic Models

Data sampling provides a way to better prepare the imbalanced training dataset prior to fitting a model. Perhaps the most popular data sampling is the SMOTE oversampling technique for creating new synthetic examples for the minority class. This can be paired with the edited nearest neighbor (ENN) algorithm that will locate and remove examples from the dataset that are ambiguous, making it easier for models to learn to discriminate between the two classes. This combination is called SMOTE-ENN and can be implemented using the `SMOTEENN` class from the `imbalanced-learn` library; for example:

```
...
# define SMOTE-ENN data sampling method
e = SMOTEENN(enn=EditedNearestNeighbours(sampling_strategy='majority'))
```

Listing 27.22: Example of defining a SMOTE-ENN sampling strategy.

SMOTE and ENN both work better when the input data is scaled beforehand. This is because both techniques involve using the nearest neighbor algorithm internally and this algorithm is sensitive to input variables with different scales. Therefore, we will require the data

to be normalized as a first step, then sampled, then used as input to the (unbalanced) logistic regression model. As such, we can use the `Pipeline` class provided by the imbalanced-learn library to create a sequence of data transforms including the data sampling method, and ending with the logistic regression model. We will compare four variations of the logistic regression model with data sampling, specifically:

- SMOTEENN + LR
- Normalization + SMOTEENN + LR
- Standardization + SMOTEENN + LR
- Normalization + Power + SMOTEENN + LR

The expectation is that LR will perform better with SMOTEENN, and that SMOTEENN will perform better with standardization or normalization. The last case does a lot, first normalizing the dataset, then applying the power transform, standardizing the result (recall that the `PowerTransformer` class will standardize the output by default), applying SMOTEENN, then finally fitting a logistic regression model. These combinations can be defined as follows:

```
# define models to test
def get_models():
    models, names = list(), list()
    # SMOTEENN
    sampling = SMOTEENN(enn=EditedNearestNeighbours(sampling_strategy='majority'))
    model = LogisticRegression(solver='liblinear')
    steps = [('e', sampling), ('m', model)]
    models.append(Pipeline(steps=steps))
    names.append('LR')
    # SMOTEENN + Norm
    sampling = SMOTEENN(enn=EditedNearestNeighbours(sampling_strategy='majority'))
    model = LogisticRegression(solver='liblinear')
    steps = [('t', MinMaxScaler()), ('e', sampling), ('m', model)]
    models.append(Pipeline(steps=steps))
    names.append('Norm')
    # SMOTEENN + Std
    sampling = SMOTEENN(enn=EditedNearestNeighbours(sampling_strategy='majority'))
    model = LogisticRegression(solver='liblinear')
    steps = [('t', StandardScaler()), ('e', sampling), ('m', model)]
    models.append(Pipeline(steps=steps))
    names.append('Std')
    # SMOTEENN + Power
    sampling = SMOTEENN(enn=EditedNearestNeighbours(sampling_strategy='majority'))
    model = LogisticRegression(solver='liblinear')
    steps = [('t1', MinMaxScaler()), ('t2', PowerTransformer()), ('e', sampling), ('m', model)]
    models.append(Pipeline(steps=steps))
    names.append('Power')
    return models, names
```

Listing 27.23: Example of a function for defining the data sampling pipelines to evaluate.

Tying this together, the complete example is listed below.

```
# compare data sampling with logistic regression on the oil spill dataset
from numpy import mean
from numpy import std
from pandas import read_csv
from matplotlib import pyplot
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.metrics import make_scorer
from sklearn.linear_model import LogisticRegression
from imblearn.metrics import geometric_mean_score
from sklearn.preprocessing import PowerTransformer
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
from imblearn.pipeline import Pipeline
from imblearn.combine import SMOTEENN
from imblearn.under_sampling import EditedNearestNeighbours

# load the dataset
def load_dataset(full_path):
    # load the dataset as a numpy array
    data = read_csv(full_path, header=None)
    # drop unused columns
    data.drop(22, axis=1, inplace=True)
    data.drop(0, axis=1, inplace=True)
    # retrieve numpy array
    data = data.values
    # split into input and output elements
    X, y = data[:, :-1], data[:, -1]
    # label encode the target variable to have the classes 0 and 1
    y = LabelEncoder().fit_transform(y)
    return X, y

# evaluate a model
def evaluate_model(X, y, model):
    # define evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # define the model evaluation the metric
    metric = make_scorer(geometric_mean_score)
    # evaluate model
    scores = cross_val_score(model, X, y, scoring=metric, cv=cv, n_jobs=-1)
    return scores

# define models to test
def get_models():
    models, names = list(), list()
    # SMOTEENN
    sampling = SMOTEENN(enn=EditedNearestNeighbours(sampling_strategy='majority'))
    model = LogisticRegression(solver='liblinear')
    steps = [('e', sampling), ('m', model)]
    models.append(Pipeline(steps=steps))
    names.append('LR')
    # SMOTEENN + Norm
    sampling = SMOTEENN(enn=EditedNearestNeighbours(sampling_strategy='majority'))
    model = LogisticRegression(solver='liblinear')
```

```

steps = [('t', MinMaxScaler()), ('e', sampling), ('m', model)]
models.append(Pipeline(steps=steps))
names.append('Norm')
# SMOTEENN + Std
sampling = SMOTEENN(enn=EditedNearestNeighbours(sampling_strategy='majority'))
model = LogisticRegression(solver='liblinear')
steps = [('t', StandardScaler()), ('e', sampling), ('m', model)]
models.append(Pipeline(steps=steps))
names.append('Std')
# SMOTEENN + Power
sampling = SMOTEENN(enn=EditedNearestNeighbours(sampling_strategy='majority'))
model = LogisticRegression(solver='liblinear')
steps = [('t1', MinMaxScaler()), ('t2', PowerTransformer()), ('e', sampling), ('m', model)]
models.append(Pipeline(steps=steps))
names.append('Power')
return models, names

# define the location of the dataset
full_path = 'oil-spill.csv'
# load the dataset
X, y = load_dataset(full_path)
# define models
models, names = get_models()
# evaluate each model
results = list()
for i in range(len(models)):
    # evaluate the model and store results
    scores = evaluate_model(X, y, models[i])
    # summarize and store
    print('>%s %.3f (%.3f)' % (names[i], mean(scores), std(scores)))
    results.append(scores)
# plot the results
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()

```

Listing 27.24: Calculate the performance of data sampling models on the oil dataset.

Running the example evaluates each version of the SMOTEENN with logistic regression model on the dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the addition of SMOTEENN improves the performance of the default LR algorithm, achieving a mean G-mean of 0.852 compared to 0.621 seen in the first set of experimental results. This is even better than balanced LR without any data scaling (previous section) that achieved a G-mean of about 0.846.

The results suggest that perhaps the final combination of normalization, power transform, and standardization achieves a slightly better score than the default LR with SMOTEENN with a G-mean of about 0.873.

```

>LR 0.852 (0.105)
>Norm 0.838 (0.130)
>Std 0.849 (0.113)

```

```
>Power 0.873 (0.118)
```

Listing 27.25: Example output from calculating the performance of data sampling models on the oil dataset.

The distribution of results can be compared with box and whisker plots. We can see the distributions all roughly have the same tight spread and that the difference in means of the results can be used to select a model.

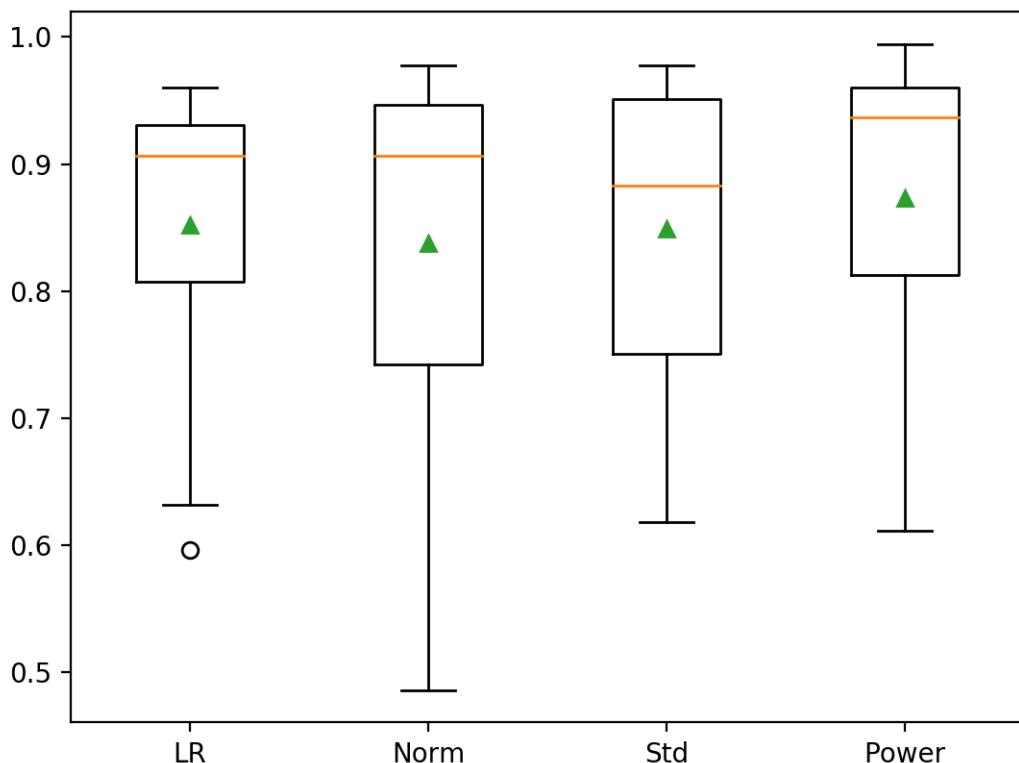


Figure 27.4: Box and Whisker Plot of Logistic Regression Models with Data Sampling on the Imbalanced Oil Spill Dataset.

27.6 Make Prediction on New Data

The use of SMOTEENN with Logistic Regression directly without any data scaling probably provides the simplest and well-performing model that could be used going forward. This model had a mean G-mean of about 0.852 on our test harness. We will use this as our final model and use it to make predictions on new data. First, we can define the model as a pipeline.

```
...
# define the model
smoteenn = SMOTEENN(enn=EditedNearestNeighbours(sampling_strategy='majority'))
model = LogisticRegression(solver='liblinear')
```

```
pipeline = Pipeline(steps=[('e', smoteenn), ('m', model)])
```

Listing 27.26: Example of defining the final model.

Once defined, we can fit it on the entire training dataset.

```
...
# fit the model
pipeline.fit(X, y)
```

Listing 27.27: Example of fitting the final model.

Once fit, we can use it to make predictions for new data by calling the `predict()` function. This will return the class label of 0 for no oil spill, or 1 for an oil spill. For example:

```
...
# define a row of data
row = [...]
# make prediction
yhat = pipeline.predict([row])
# get the label
label = yhat[0]
```

Listing 27.28: Example of making a prediction with the fit final model.

To demonstrate this, we can use the fit model to make some predictions of labels for a few cases where we know there is no oil spill, and a few where we know there is. The complete example is listed below.

```
# fit a model and make predictions for the oil spill dataset
from pandas import read_csv
from sklearn.preprocessing import LabelEncoder
from sklearn.linear_model import LogisticRegression
from imblearn.pipeline import Pipeline
from imblearn.combine import SMOTEENN
from imblearn.under_sampling import EditedNearestNeighbours

# load the dataset
def load_dataset(full_path):
    # load the dataset as a numpy array
    data = read_csv(full_path, header=None)
    # retrieve numpy array
    data = data.values
    # split into input and output elements
    X, y = data[:, 1:-1], data[:, -1]
    # label encode the target variable to have the classes 0 and 1
    y = LabelEncoder().fit_transform(y)
    return X, y

# define the location of the dataset
full_path = 'oil-spill.csv'
# load the dataset
X, y = load_dataset(full_path)
# define the model
smoteenn = SMOTEENN(enn=EditedNearestNeighbours(sampling_strategy='majority'))
model = LogisticRegression(solver='liblinear')
pipeline = Pipeline(steps=[('e', smoteenn), ('m', model)])
# fit the model
```

```

pipeline.fit(X, y)
# evaluate on some non-spill cases (known class 0)
print('Non-Spill Cases:')
data = [[329, 1627.54, 1409.43, 51, 822500, 35, 6.1, 4610, 0.17, 178.4, 0.2, 0.24, 0.39,
         0.12, 0.27, 138.32, 34.81, 2.02, 0.14, 0.19, 75.26, 0, 0.47, 351.67, 0.18, 9.24, 0.38,
         2.57, -2.96, -0.28, 1.93, 0, 1.93, 34, 1710, 0, 25.84, 78, 55, 1460.31, 710.63, 451.78,
         150.85, 3.23, 0, 4530.75, 66.25, 7.85],
[3234, 1091.56, 1357.96, 32, 8085000, 40.08, 8.98, 25450, 0.22, 317.7, 0.18, 0.2, 0.49,
         0.09, 0.41, 114.69, 41.87, 2.31, 0.15, 0.18, 75.26, 0, 0.53, 351.67, 0.18, 9.24,
         0.24, 3.56, -3.09, -0.31, 2.17, 0, 2.17, 281, 14490, 0, 80.11, 78, 55, 4287.77,
         3095.56, 1937.42, 773.69, 2.21, 0, 4927.51, 66.15, 7.24],
[2339, 1537.68, 1633.02, 45, 5847500, 38.13, 9.29, 22110, 0.24, 264.5, 0.21, 0.26, 0.79,
         0.08, 0.71, 89.49, 32.23, 2.2, 0.17, 0.22, 75.26, 0, 0.51, 351.67, 0.18, 9.24, 0.27,
         4.21, -2.84, -0.29, 2.16, 0, 2.16, 228, 12150, 0, 83.6, 78, 55, 3959.8, 2404.16,
         1530.38, 659.67, 2.59, 0, 4732.04, 66.34, 7.67]]
for row in data:
    # make prediction
    yhat = pipeline.predict([row])
    # get the label
    label = yhat[0]
    # summarize
    print('>Predicted=%d (expected 0)' % (label))
# evaluate on some spill cases (known class 1)
print('Spill Cases:')
data = [[2971, 1020.91, 630.8, 59, 7427500, 32.76, 10.48, 17380, 0.32, 427.4, 0.22, 0.29,
         0.5, 0.08, 0.42, 149.87, 50.99, 1.89, 0.14, 0.18, 75.26, 0, 0.44, 351.67, 0.18, 9.24,
         2.5, 10.63, -3.07, -0.28, 2.18, 0, 2.18, 164, 8730, 0, 40.67, 78, 55, 5650.88, 1749.29,
         1245.07, 348.7, 4.54, 0, 25579.34, 65.78, 7.41],
[3155, 1118.08, 469.39, 11, 7887500, 30.41, 7.99, 15880, 0.26, 496.7, 0.2, 0.26, 0.69,
         0.11, 0.58, 118.11, 43.96, 1.76, 0.15, 0.18, 75.26, 0, 0.4, 351.67, 0.18, 9.24, 0.78,
         8.68, -3.19, -0.33, 2.19, 0, 2.19, 150, 8100, 0, 31.97, 78, 55, 3471.31, 3059.41,
         2043.9, 477.23, 1.7, 0, 28172.07, 65.72, 7.58],
[115, 1449.85, 608.43, 88, 287500, 40.42, 7.34, 3340, 0.18, 86.1, 0.21, 0.32, 0.5, 0.17,
         0.34, 71.2, 16.73, 1.82, 0.19, 0.29, 87.65, 0, 0.46, 132.78, -0.01, 3.78, 0.7, 4.79,
         -3.36, -0.23, 1.95, 0, 1.95, 29, 1530, 0.01, 38.8, 89, 69, 1400, 250, 150, 45.13,
         9.33, 1, 31692.84, 65.81, 7.84]]
for row in data:
    # make prediction
    yhat = pipeline.predict([row])
    # get the label
    label = yhat[0]
    # summarize
    print('>Predicted=%d (expected 1)' % (label))

```

Listing 27.29: Fit a model and use it to make predictions.

Running the example first fits the model on the entire training dataset. Then the fit model used to predict the label of an oil spill for cases where we know there is none, chosen from the dataset file. We can see that all cases are correctly predicted. Then some cases of actual oil spills are used as input to the model and the label is predicted. As we might have hoped, the correct labels are again predicted.

```

Non-Spill Cases:
>Predicted=0 (expected 0)
>Predicted=0 (expected 0)
>Predicted=0 (expected 0)

```

```
Spill Cases:  
>Predicted=1 (expected 1)  
>Predicted=1 (expected 1)  
>Predicted=1 (expected 1)
```

Listing 27.30: Example output from fitting a model and use it to make predictions.

27.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

27.7.1 Papers

- *Machine Learning for the Detection of Oil Spills in Satellite Radar Images*, 1998.
<https://link.springer.com/article/10.1023/A:1007452223027>

27.7.2 APIs

- pandas.read_csv API.
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html
- sklearn.dummy.DummyClassifier API.
<https://scikit-learn.org/stable/modules/generated/sklearn.dummy.DummyClassifier.html>
- imblearn.metrics.geometric_mean_score API.
https://imbalanced-learn.org/stable/generated/imblearn.metrics.geometric_mean_score.html
- sklearn.pipeline.Pipeline API.
<https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html>
- sklearn.linear_model.LogisticRegression API.
https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
- imblearn.combine.SMOTEEENN API.
<https://imbalanced-learn.org/stable/generated/imblearn.combine.SMOTEEENN.html>

27.7.3 Dataset

- Oil Spill Dataset CSV.
<https://raw.githubusercontent.com/jbrownlee/Datasets/master/oil-spill.csv>
- Oil Spill Dataset Details.
<https://raw.githubusercontent.com/jbrownlee/Datasets/master/oil-spill.names>

27.8 Summary

In this tutorial, you discovered how to develop a model to predict the presence of an oil spill in satellite images and evaluate it using the G-mean metric. Specifically, you learned:

- How to load and explore the dataset and generate ideas for data preparation and model selection.
- How to evaluate a suite of probabilistic models and improve their performance with appropriate data preparation.
- How to fit a final model and use it to predict class labels for specific cases.

27.8.1 Next

In the next tutorial, you will discover how to systematically work through the German credit imbalanced classification dataset.

Chapter 28

Project: German Credit Classification

Misclassification errors on the minority class are more important than other types of prediction errors for some imbalanced classification tasks. One example is the problem of classifying bank customers as to whether they should receive a loan or not. Giving a loan to a bad customer marked as a good customer results in a greater cost to the bank than denying a loan to a good customer marked as a bad customer. This requires careful selection of a performance metric that both promotes minimizing misclassification errors in general, and favors minimizing one type of misclassification error over another.

The German credit dataset is a standard imbalanced classification dataset that has this property of differing costs to misclassification errors. Models evaluated on this dataset can be evaluated using the Fbeta-measure that provides a way of both quantifying model performance generally, and captures the requirement that one type of misclassification error is more costly than another. In this tutorial, you will discover how to develop and evaluate a model for the imbalanced German credit classification dataset. After completing this tutorial, you will know:

- How to load and explore the dataset and generate ideas for data preparation and model selection.
- How to evaluate a suite of machine learning models and improve their performance with undersampling techniques.
- How to fit a final model and use it to predict class labels for specific cases.

Let's get started.

Note: This chapter makes use of the imbalanced-learn library. See Appendix [B](#) for installation instructions, if needed.

28.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. German Credit Dataset
2. Explore the Dataset

3. Model Test and Baseline Result
4. Evaluate Models
5. Make Prediction on New Data

28.2 German Credit Dataset

In this project, we will use a standard imbalanced machine learning dataset referred to as the *German Credit* dataset or simply *German*. The dataset was used as part of the Statlog project, a European-based initiative in the 1990s to evaluate and compare a large number (at the time) of machine learning algorithms on a range of different classification tasks. The dataset is credited to Hans Hofmann.

The fragmentation amongst different disciplines has almost certainly hindered communication and progress. The StatLog project was designed to break down these divisions by selecting classification procedures regardless of historical pedigree, testing them on large-scale and commercially important problems, and hence to determine to what extent the various techniques met the needs of industry.

— Page 4, *Machine Learning, Neural and Statistical Classification*, 1994.

The German credit dataset describes financial and banking details for customers and the task is to determine whether the customer is good or bad. The assumption is that the task involves predicting whether a customer will pay back a loan or credit. The dataset includes 1,000 examples and 20 input variables, 7 of which are numerical (integer) and 13 are categorical.

- Status of existing checking account
- Duration in month
- Credit history
- Purpose
- Credit amount
- Savings account
- Present employment since
- Installment rate in percentage of disposable income
- Personal status and sex
- Other debtors
- Present residence since
- Property

- Age in years
- Other installment plans
- Housing
- Number of existing credits at this bank
- Job
- Number of dependents
- Telephone
- Foreign worker

Some of the categorical variables have an ordinal relationship, such as *Savings account*, although most do not. There are two outcome classes, 1 for good customers and 2 for bad customers. Good customers are the default or negative class, whereas bad customers are the exception or positive class. A total of 70 percent of the examples are good customers, whereas the remaining 30 percent of examples are bad customers.

- **Good Customers:** Negative or majority class (70%).
- **Bad Customers:** Positive or minority class (30%).

A cost matrix is provided with the dataset that gives a different penalty to each misclassification error for the positive class. Specifically, a cost of five is applied to a false negative (marking a bad customer as good) and a cost of one is assigned for a false positive (marking a good customer as bad).

- **Cost for False Negative:** 5
- **Cost for False Positive:** 1

This suggests that the positive class is the focus of the prediction task and that it is more costly to the bank or financial institution to give money to a bad customer than to not give money to a good customer. This must be taken into account when selecting a performance metric. Next, let's take a closer look at the data.

28.3 Explore the Dataset

First, download the dataset and save it in your current working directory with the name `german.csv`.

- Download German Credit Dataset (`german.csv`). ¹

Review the contents of the file. The first few lines of the file should look as follows:

¹<https://raw.githubusercontent.com/jbrownlee/Datasets/master/german.csv>

```
A11,6,A34,A43,1169,A65,A75,4,A93,A101,4,A121,67,A143,A152,2,A173,1,A192,A201,1
A12,48,A32,A43,5951,A61,A73,2,A92,A101,2,A121,22,A143,A152,1,A173,1,A191,A201,2
A14,12,A34,A46,2096,A61,A74,2,A93,A101,3,A121,49,A143,A152,1,A172,2,A191,A201,1
A11,42,A32,A42,7882,A61,A74,2,A93,A103,4,A122,45,A143,A153,1,A173,2,A191,A201,1
A11,24,A33,A40,4870,A61,A73,3,A93,A101,4,A124,53,A143,A153,2,A173,2,A191,A201,2
...

```

Listing 28.1: Example of rows of data from the German dataset.

We can see that the categorical columns are encoded with an `Axxx` format, where `xxxx` are integers for different labels. A one hot encoding of the categorical variables will be required. We can also see that the numerical variables have different scales, e.g. 6, 48, and 12 in column 2, and 1169, 5951, etc. in column 5. This suggests that scaling of the integer columns will be needed for those algorithms that are sensitive to scale.

The target variable or class is the last column and contains values of 1 and 2. These will need to be label encoded to 0 and 1, respectively, to meet the general expectation for imbalanced binary classification tasks where 0 represents the negative case and 1 represents the positive case. The dataset can be loaded as a `DataFrame` using the `read_csv()` Pandas function, specifying the location and the fact that there is no header line.

```
...
# define the dataset location
filename = 'german.csv'
# load the csv file as a data frame
dataframe = read_csv(filename, header=None)
```

Listing 28.2: Example of loading the German dataset.

Once loaded, we can summarize the number of rows and columns by printing the shape of the `DataFrame`.

```
...
# summarize the shape of the dataset
print(dataframe.shape)
```

Listing 28.3: Example of summarizing the shape of the loaded dataset.

We can also summarize the number of examples in each class using the `Counter` object.

```
...
# summarize the class distribution
target = dataframe.values[:, -1]
counter = Counter(target)
for k, v in counter.items():
    per = v / len(target) * 100
    print('Class=%d, Count=%d, Percentage=%.3f%%' % (k, v, per))
```

Listing 28.4: Example of summarizing the class distribution for the loaded dataset.

Tying this together, the complete example of loading and summarizing the dataset is listed below.

```
# load and summarize the dataset
from pandas import read_csv
from collections import Counter
# define the dataset location
```

```

filename = 'german.csv'
# load the csv file as a data frame
dataframe = read_csv(filename, header=None)
# summarize the shape of the dataset
print(dataframe.shape)
# summarize the class distribution
target = dataframe.values[:, -1]
counter = Counter(target)
for k, v in counter.items():
    per = v / len(target) * 100
    print('Class=%d, Count=%d, Percentage=%.3f%%' % (k, v, per))

```

Listing 28.5: Load and summarize the German dataset.

Running the example first loads the dataset and confirms the number of rows and columns, that is 1,000 rows and 20 input variables and 1 target variable. The class distribution is then summarized, confirming the number of good and bad customers and the percentage of cases in the minority and majority classes.

```
(1000, 21)
Class=1, Count=700, Percentage=70.000%
Class=2, Count=300, Percentage=30.000%
```

Listing 28.6: Example output from loading and summarizing the German dataset.

We can also take a look at the distribution of the seven numerical input variables by creating a histogram for each. First, we can select the columns with numeric variables by calling the `select_dtypes()` function on the `DataFrame`. We can then select just those columns from the `DataFrame`. We would expect there to be seven, plus the numerical class labels.

```

...
# select columns with numerical data types
num_ix = df.select_dtypes(include=['int64', 'float64']).columns
# select a subset of the dataframe with the chosen columns
subset = df[num_ix]
```

Listing 28.7: Example of selecting the numerical attributes.

We can then create histograms of each numeric input variable. The complete example is listed below.

```

# create histograms of numeric input variables
from pandas import read_csv
from matplotlib import pyplot
# define the dataset location
filename = 'german.csv'
# load the csv file as a data frame
df = read_csv(filename, header=None)
# select columns with numerical data types
num_ix = df.select_dtypes(include=['int64', 'float64']).columns
# select a subset of the dataframe with the chosen columns
subset = df[num_ix]
# create a histogram plot of each numeric variable
ax = subset.hist()
# disable axis labels to avoid the clutter
for axis in ax.flatten():
    axis.set_xticklabels([])
```

```

axis.set_yticklabels([])
# show the plot
pyplot.show()

```

Listing 28.8: Plot histograms of the German dataset.

Running the example creates the figure with one histogram subplot for each of the seven numeric input variables and one class label in the dataset. The title of each subplot indicates the column number in the `DataFrame` (e.g. zero-offset from 0 to 20). We can see many different distributions, some with Gaussian-like distributions, others with seemingly exponential or discrete distributions. Depending on the choice of modeling algorithms, we would expect scaling the distributions to the same range to be useful, and perhaps the use of some power transforms.

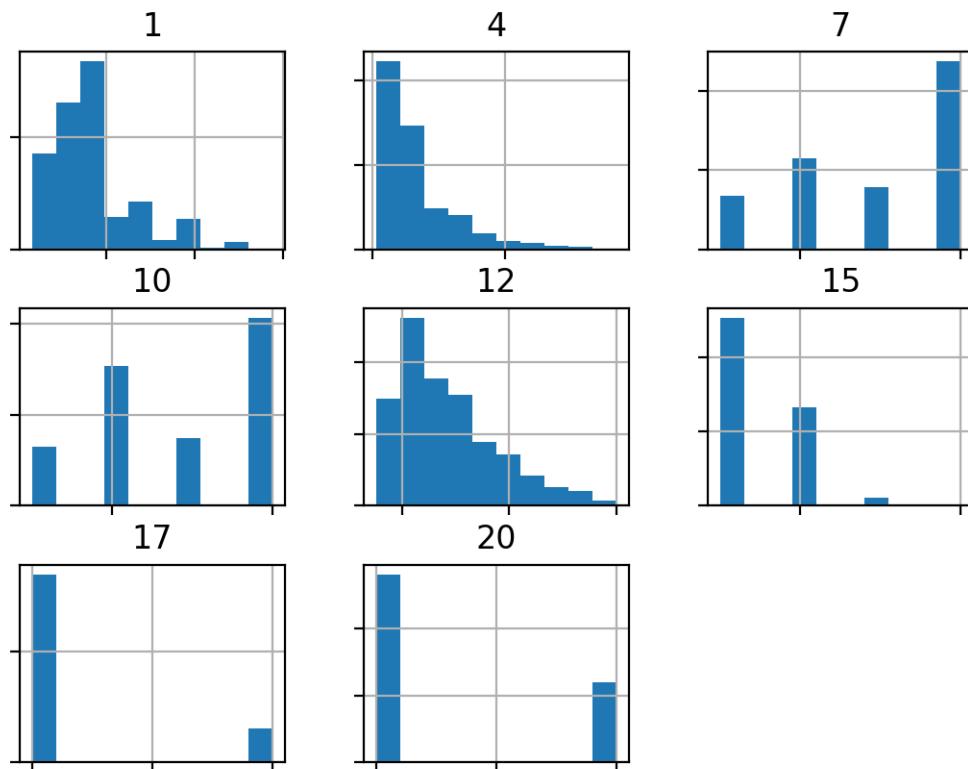


Figure 28.1: Histogram of Numeric Variables in the German Credit Dataset.

Now that we have reviewed the dataset, let's look at developing a test harness for evaluating candidate models.

28.4 Model Test and Baseline Result

We will evaluate candidate models using repeated stratified k -fold cross-validation. The k -fold cross-validation procedure provides a good general estimate of model performance that is not

too optimistically biased, at least compared to a single train-test split. We will use $k=10$, meaning each fold will contain about $\frac{1000}{10}$ or 100 examples. Stratified means that each fold will contain the same mixture of examples by class, that is about 70 percent to 30 percent good to bad customers. Repeated means that the evaluation process will be performed multiple times to help avoid fluke results and better capture the variance of the chosen model. We will use three repeats. This means a single model will be fit and evaluated 10×3 (30) times and the mean and standard deviation of these runs will be reported. This can be achieved by using the `RepeatedStratifiedKFold` scikit-learn class.

We will predict class labels of whether a customer is good or not. Therefore, we need a measure that is appropriate for evaluating the predicted class labels. The focus of the task is on the positive class (bad customers). Precision and recall are a good place to start. Maximizing precision will minimize the false positives and maximizing recall will minimize the false negatives in the predictions made by a model.

Using the F-measure will calculate the harmonic mean between precision and recall (for more on the F-measure, see Chapter 6). This is a good single number that can be used to compare and select a model on this problem. The issue is that false negatives are more damaging than false positives. Remember that false negatives on this dataset are cases of a bad customer being marked as a good customer and being given a loan.

- **False Negative:** Bad Customer (class 1) predicted as a Good Customer (class 0).
- **False Positive:** Good Customer (class 0) predicted as a Bad Customer (class 1).

False positives are cases of a good customer being marked as a bad customer and not being given a loan. False negatives are more costly to the bank than false positives.

$$Cost(\text{FalseNegatives}) > Cost(\text{FalsePositives}) \quad (28.1)$$

Put another way, we are interested in the F-measure that will summarize a model's ability to minimize misclassification errors for the positive class, but we want to favor models that are better at minimizing false negatives over false positives. This can be achieved by using a version of the F-measure that calculates a weighted harmonic mean of precision and recall but favors higher recall scores over precision scores. This is called the F β -measure, a generalization of F-measure, where β is a parameter that defines the weighting of the two scores. A β value of 2 will weight more attention on recall than precision and is referred to as the F2-measure.

$$\text{F2-measure} = \frac{(1 + 2^2) \times \text{Precision} \times \text{Recall}}{2^2 \times \text{Precision} + \text{Recall}} \quad (28.2)$$

We will use this measure to evaluate models on the German credit dataset. This can be achieved using the `fbeta_score()` scikit-learn function. We can define a function to load the dataset and split the columns into input and output variables. We will one hot encode the categorical variables and label encode the target variable. You might recall that a one hot encoding replaces the categorical variable with one new column for each value of the variable and marks values with a 1 in the column for that value. First, we must split the `DataFrame` into input and output variables.

```
...
# split into inputs and outputs
last_ix = len(dataframe.columns) - 1
```

```
X, y = dataframe.drop(last_ix, axis=1), dataframe[last_ix]
```

Listing 28.9: Example of splitting the dataset into input and output components.

Next, we need to select all input variables that are categorical, then apply a one hot encoding and leave the numerical variables untouched. This can be achieved using a `ColumnTransformer` and defining the transform as a `OneHotEncoder` applied only to the column indices for categorical variables.

```
...
# select categorical features
cat_ix = X.select_dtypes(include=['object', 'bool']).columns
# one hot encode cat features only
ct = ColumnTransformer([('o', OneHotEncoder(), cat_ix)], remainder='passthrough')
X = ct.fit_transform(X)
```

Listing 28.10: Example of encoding the categorical variables.

We can then label encode the target variable.

```
...
# label encode the target variable to have the classes 0 and 1
y = LabelEncoder().fit_transform(y)
```

Listing 28.11: Example of label encoding the target variable.

The `load_dataset()` function below ties all of this together and loads and prepares the dataset for modeling.

```
# load the dataset
def load_dataset(full_path):
    # load the dataset as a numpy array
    dataframe = read_csv(full_path, header=None)
    # split into inputs and outputs
    last_ix = len(dataframe.columns) - 1
    X, y = dataframe.drop(last_ix, axis=1), dataframe[last_ix]
    # select categorical features
    cat_ix = X.select_dtypes(include=['object', 'bool']).columns
    # one hot encode cat features only
    ct = ColumnTransformer([('o', OneHotEncoder(), cat_ix)], remainder='passthrough')
    X = ct.fit_transform(X)
    # label encode the target variable to have the classes 0 and 1
    y = LabelEncoder().fit_transform(y)
    return X, y
```

Listing 28.12: Example of a function for loading and preparing the dataset for modeling.

Next, we need a function that will evaluate a set of predictions using the `fbeta_score()` function with beta set to 2.

```
# calculate f2-measure
def f2_measure(y_true, y_pred):
    return fbeta_score(y_true, y_pred, beta=2)
```

Listing 28.13: Example of a function for calculating the F2-measure.

We can then define a function that will evaluate a given model on the dataset and return a list of F2-measure scores for each fold and repeat. The `evaluate_model()` function below implements this, taking the dataset and model as arguments and returning the list of scores.

```
# evaluate a model
def evaluate_model(X, y, model):
    # define evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # define the model evaluation metric
    metric = make_scorer(f2_measure)
    # evaluate model
    scores = cross_val_score(model, X, y, scoring=metric, cv=cv, n_jobs=-1)
    return scores
```

Listing 28.14: Example of a function for evaluating a defined model.

Finally, we can evaluate a baseline model on the dataset using this test harness. A model that predicts the minority class for examples will achieve a maximum recall score and a baseline precision score. The F2-measure from this prediction provides a baseline in model performance on this problem by which all other models can be compared. This can be achieved using the `DummyClassifier` class from the scikit-learn library and setting the `strategy` argument to ‘constant’ and the `constant` argument to 1 for the minority class.

```
...
# define the reference model
model = DummyClassifier(strategy='constant', constant=1)
```

Listing 28.15: Example of defining the baseline model.

Once the model is evaluated, we can report the mean and standard deviation of the F2-measure scores directly.

```
...
# evaluate the model
scores = evaluate_model(X, y, model)
# summarize performance
print('Mean F2: %.3f (%.3f)' % (mean(scores), std(scores)))
```

Listing 28.16: Example of evaluating the model and summarizing the performance.

Tying this together, the complete example of loading the German Credit dataset, evaluating a baseline model, and reporting the performance is listed below.

```
# test harness and baseline model evaluation for the german credit dataset
from collections import Counter
from numpy import mean
from numpy import std
from pandas import read_csv
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.metrics import fbeta_score
from sklearn.metrics import make_scorer
from sklearn.dummy import DummyClassifier

# load the dataset
def load_dataset(full_path):
    # load the dataset as a numpy array
```

```

dataframe = read_csv(full_path, header=None)
# split into inputs and outputs
last_ix = len(dataframe.columns) - 1
X, y = dataframe.drop(last_ix, axis=1), dataframe[last_ix]
# select categorical features
cat_ix = X.select_dtypes(include=['object', 'bool']).columns
# one hot encode cat features only
ct = ColumnTransformer([('o', OneHotEncoder(), cat_ix)], remainder='passthrough')
X = ct.fit_transform(X)
# label encode the target variable to have the classes 0 and 1
y = LabelEncoder().fit_transform(y)
return X, y

# calculate f2-measure
def f2_measure(y_true, y_pred):
    return fbeta_score(y_true, y_pred, beta=2)

# evaluate a model
def evaluate_model(X, y, model):
    # define evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # define the model evaluation metric
    metric = make_scorer(f2_measure)
    # evaluate model
    scores = cross_val_score(model, X, y, scoring=metric, cv=cv, n_jobs=-1)
    return scores

# define the location of the dataset
full_path = 'german.csv'
# load the dataset
X, y = load_dataset(full_path)
# summarize the loaded dataset
print(X.shape, y.shape, Counter(y))
# define the reference model
model = DummyClassifier(strategy='constant', constant=1)
# evaluate the model
scores = evaluate_model(X, y, model)
# summarize performance
print('Mean F2: %.3f (%.3f)' % (mean(scores), std(scores)))

```

Listing 28.17: Calculate the baseline performance for the German dataset.

Running the example first loads and summarizes the dataset. We can see that we have the correct number of rows loaded, and through the one hot encoding of the categorical input variables, we have increased the number of input variables from 20 to 61. That suggests that the 13 categorical variables were encoded into a total of 54 columns.

Importantly, we can see that the class labels have the correct mapping to integers with 0 for the majority class and 1 for the minority class, customary for imbalanced binary classification dataset. Next, the average of the F2-measure scores is reported. In this case, we can see that the baseline algorithm achieves an F2-measure of about 0.682. This score provides a lower limit on model skill; any model that achieves an average F2-measure above about 0.682 has skill, whereas models that achieve a score below this value do not have skill on this dataset.

```
(1000, 61) (1000,) Counter({0: 700, 1: 300})
Mean F2: 0.682 (0.000)
```

Listing 28.18: Example output from calculating the baseline performance for the German dataset.

Now that we have a test harness and a baseline in performance, we can begin to evaluate some models on this dataset.

28.5 Evaluate Models

In this section, we will evaluate a suite of different techniques on the dataset using the test harness developed in the previous section. The goal is to both demonstrate how to work through the problem systematically and to demonstrate the capability of some techniques designed for imbalanced classification problems. The reported performance is good, but not highly optimized (e.g. hyperparameters are not tuned).

28.5.1 Evaluate Machine Learning Algorithms

Let's start by evaluating a mixture of probabilistic machine learning models on the dataset. It can be a good idea to spot-check a suite of different linear and nonlinear algorithms on a dataset to quickly flush out what works well and deserves further attention, and what doesn't. We will evaluate the following machine learning models on the German credit dataset:

- Logistic Regression (LR)
- Linear Discriminant Analysis (LDA)
- Naive Bayes (NB)
- Gaussian Process Classifier (GPC)
- Support Vector Machine (SVM)

We will use mostly default model hyperparameters. We will define each model in turn and add them to a list so that we can evaluate them sequentially. The `get_models()` function below defines the list of models for evaluation, as well as a list of model short names for plotting the results later.

```
# define models to test
def get_models():
    models, names = list(), list()
    # LR
    models.append(LogisticRegression(solver='liblinear'))
    names.append('LR')
    # LDA
    models.append(LinearDiscriminantAnalysis())
    names.append('LDA')
    # NB
    models.append(GaussianNB())
    names.append('NB')
    # GPC
    models.append(GaussianProcessClassifier())
```

```

names.append('GPC')
# SVM
models.append(SVC(gamma='scale'))
names.append('SVM')
return models, names

```

Listing 28.19: Example of a function for defining the models to evaluate.

We can then enumerate the list of models in turn and evaluate each, storing the scores for later evaluation. We will one hot encode the categorical input variables as we did in the previous section, and in this case, we will normalize the numerical input variables. This is best performed using the `MinMaxScaler` within each fold of the cross-validation evaluation process.

An easy way to implement this is to use a `Pipeline` where the first step is a `ColumnTransformer` that applies a `OneHotEncoder` to just the categorical variables, and a `MinMaxScaler` to just the numerical input variables. To achieve this, we need a list of the column indices for categorical and numerical input variables. We can update the `load_dataset()` to return the column indexes as well as the input and output elements of the dataset. The updated version of this function is listed below.

```

# load the dataset
def load_dataset(full_path):
    # load the dataset as a numpy array
    dataframe = read_csv(full_path, header=None)
    # split into inputs and outputs
    last_ix = len(dataframe.columns) - 1
    X, y = dataframe.drop(last_ix, axis=1), dataframe[last_ix]
    # select categorical and numerical features
    cat_ix = X.select_dtypes(include=['object', 'bool']).columns
    num_ix = X.select_dtypes(include=['int64', 'float64']).columns
    # label encode the target variable to have the classes 0 and 1
    y = LabelEncoder().fit_transform(y)
    return X.values, y, cat_ix, num_ix

```

Listing 28.20: Example of a function for loading and preparing the dataset.

We can then call this function to get the data and the list of categorical and numerical variables.

```

...
# define the location of the dataset
full_path = 'german.csv'
# load the dataset
X, y, cat_ix, num_ix = load_dataset(full_path)

```

Listing 28.21: Example of loading the dataset.

This can be used to prepare a `Pipeline` to wrap each model prior to evaluating it. First, the `ColumnTransformer` is defined, which specifies what transform to apply to each type of column, then this is used as the first step in a `Pipeline` that ends with the specific model that will be fit and evaluated.

```

...
# evaluate each model
for i in range(len(models)):
    # one hot encode categorical, normalize numerical
    ct = ColumnTransformer([('c', OneHotEncoder(), cat_ix), ('n', MinMaxScaler(), num_ix)])

```

```
# wrap the model in a pipeline
pipeline = Pipeline(steps=[('t',ct),('m',models[i])])
# evaluate the model and store results
scores = evaluate_model(X, y, pipeline)
```

Listing 28.22: Example of evaluating the defined models.

We can summarize the mean F2-measure for each algorithm; this will help to directly compare algorithms.

```
...
# summarize and store
print('>%s %.3f (%.3f)' % (names[i], mean(scores), std(scores)))
```

Listing 28.23: Example of reporting the performance of an evaluated model.

At the end of the run, we will create a separate box and whisker plot for each algorithm's sample of results. These plots will use the same y-axis scale so we can compare the distribution of results directly.

```
...
# plot the results
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 28.24: Example of summarizing performance using box and whisker plots.

Tying this all together, the complete example of evaluating a suite of machine learning algorithms on the German credit dataset is listed below.

```
# spot check machine learning algorithms on the german credit dataset
from numpy import mean
from numpy import std
from pandas import read_csv
from matplotlib import pyplot
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import MinMaxScaler
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.metrics import fbeta_score
from sklearn.metrics import make_scorer
from sklearn.linear_model import LogisticRegression
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.svm import SVC

# load the dataset
def load_dataset(full_path):
    # load the dataset as a numpy array
    dataframe = read_csv(full_path, header=None)
    # split into inputs and outputs
    last_ix = len(dataframe.columns) - 1
    X, y = dataframe.drop(last_ix, axis=1), dataframe[last_ix]
```

```
# select categorical and numerical features
cat_ix = X.select_dtypes(include=['object', 'bool']).columns
num_ix = X.select_dtypes(include=['int64', 'float64']).columns
# label encode the target variable to have the classes 0 and 1
y = LabelEncoder().fit_transform(y)
return X.values, y, cat_ix, num_ix

# calculate f2-measure
def f2_measure(y_true, y_pred):
    return fbeta_score(y_true, y_pred, beta=2)

# evaluate a model
def evaluate_model(X, y, model):
    # define evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # define the model evaluation metric
    metric = make_scorer(f2_measure)
    # evaluate model
    scores = cross_val_score(model, X, y, scoring=metric, cv=cv, n_jobs=-1)
    return scores

# define models to test
def get_models():
    models, names = list(), list()
    # LR
    models.append(LogisticRegression(solver='liblinear'))
    names.append('LR')
    # LDA
    models.append(LinearDiscriminantAnalysis())
    names.append('LDA')
    # NB
    models.append(GaussianNB())
    names.append('NB')
    # GPC
    models.append(GaussianProcessClassifier())
    names.append('GPC')
    # SVM
    models.append(SVC(gamma='scale'))
    names.append('SVM')
    return models, names

# define the location of the dataset
full_path = 'german.csv'
# load the dataset
X, y, cat_ix, num_ix = load_dataset(full_path)
# define models
models, names = get_models()
results = list()
# evaluate each model
for i in range(len(models)):
    # one hot encode categorical, normalize numerical
    ct = ColumnTransformer([('c', OneHotEncoder(), cat_ix), ('n', MinMaxScaler(), num_ix)])
    # wrap the model in a pipeline
    pipeline = Pipeline(steps=[('t', ct), ('m', models[i])])
    # evaluate the model and store results
    scores = evaluate_model(X, y, pipeline)
    results.append(scores)
```

```
results.append(scores)
# summarize and store
print('>%s %.3f (%.3f)' % (names[i], mean(scores), std(scores)))
# plot the results
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 28.25: Calculate the performance of machine learning algorithms for the German dataset.

Running the example evaluates each algorithm in turn and reports the mean and standard deviation F2-measure.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that none of the tested models have an F2-measure above the default of predicting the majority class in all cases (0.682). None of the models are skillful. This is surprising, although suggests that perhaps the decision boundary between the two classes is noisy.

```
>LR 0.497 (0.072)
>LDA 0.519 (0.072)
>NB 0.639 (0.049)
>GPC 0.219 (0.061)
>SVM 0.436 (0.077)
```

Listing 28.26: Example output from calculating the performance of machine learning models on the German dataset.

A figure is created showing one box and whisker plot for each algorithm's sample of results. The box shows the middle 50 percent of the data, the orange line in the middle of each box shows the median of the sample, and the green triangle in each box shows the mean of the sample.

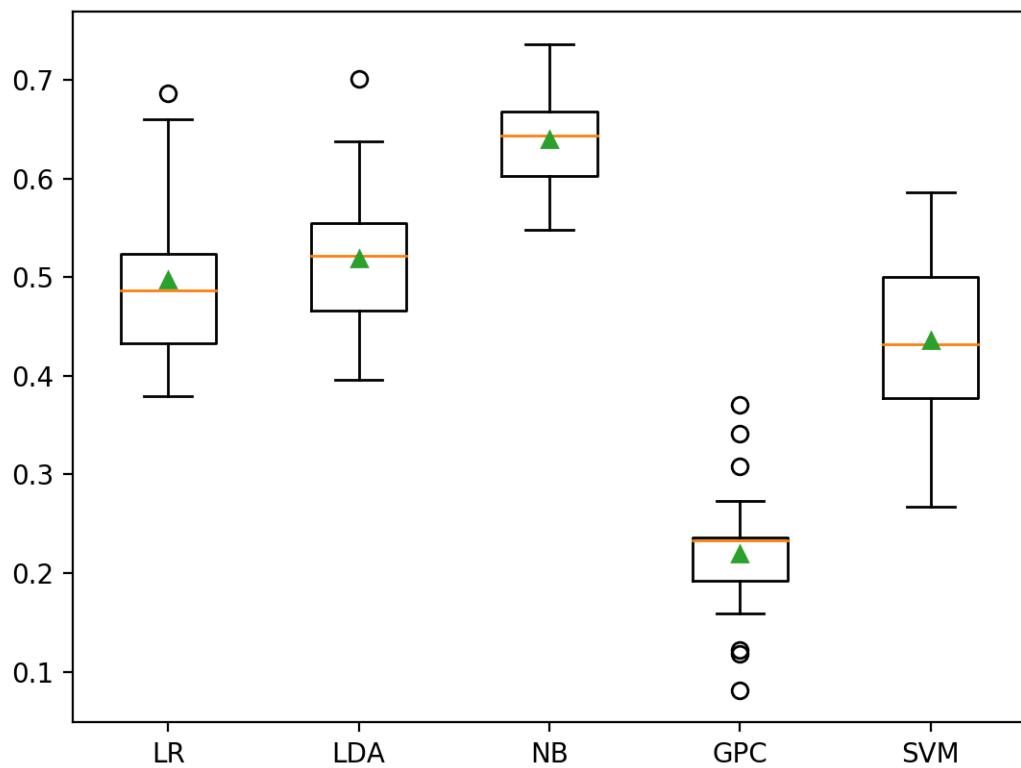


Figure 28.2: Box and Whisker Plot of Machine Learning Models on the Imbalanced German Credit Dataset.

Now that we have some results, let's see if we can improve them with some undersampling.

28.5.2 Evaluate Undersampling

Undersampling is perhaps the least widely used technique when addressing an imbalanced classification task as most of the focus is put on oversampling the majority class with SMOTE. Undersampling can help to remove examples from the majority class along the decision boundary that make the problem challenging for classification algorithms. In this experiment we will test the following undersampling algorithms:

- Tomek Links (TL)
- Edited Nearest Neighbors (ENN)
- Repeated Edited Nearest Neighbors (RENN)
- One Sided Selection (OSS)
- Neighborhood Cleaning Rule (NCR)

The Tomek Links and ENN methods select examples from the majority class to delete, whereas OSS and NCR both select examples to keep and examples to delete. We will use the cost-sensitive version of the logistic regression algorithm to test each undersampling method in an effort to further lift model performance. The `get_models()` function from the previous section can be updated to return a list of undersampling techniques to test with the logistic regression algorithm. We use the implementations of these algorithms from the imbalanced-learn library. The updated version of the `get_models()` function defining the undersampling methods is listed below.

```
# define undersampling models to test
def get_models():
    models, names = list(), list()
    # TL
    models.append(TomekLinks())
    names.append('TL')
    # ENN
    models.append(EditedNearestNeighbours())
    names.append('ENN')
    # RENN
    models.append(RepeatedEditedNearestNeighbours())
    names.append('RENN')
    # OSS
    models.append(OneSidedSelection())
    names.append('OSS')
    # NCR
    models.append(NeighbourhoodCleaningRule())
    names.append('NCR')
    return models, names
```

Listing 28.27: Example of defining undersampling algorithms to evaluate.

The `Pipeline` provided by scikit-learn does not know about undersampling algorithms. Therefore, we must use the `Pipeline` implementation provided by the imbalanced-learn library. As in the previous section, the first step of the pipeline will be one hot encoding of categorical variables and normalization of numerical variables, and the final step will be fitting the model. Here, the middle step will be the undersampling technique, correctly applied within the cross-validation evaluation on the training dataset only.

```
...
# define model to evaluate
model = LogisticRegression(solver='liblinear')
# one hot encode categorical, normalize numerical
ct = ColumnTransformer([('c',OneHotEncoder(),cat_ix), ('n',MinMaxScaler(),num_ix)])
# scale, then undersample, then fit model
pipeline = Pipeline(steps=[('t',ct), ('s', models[i]), ('m',model)])
# evaluate the model and store results
scores = evaluate_model(X, y, pipeline)
```

Listing 28.28: Example of defining a pipeline for a data sampling algorithm.

Tying this together, the complete example of evaluating logistic regression with different undersampling methods on the German credit dataset is listed below. We would expect the undersampling to result in a lift on skill in logistic regression, ideally above the baseline performance of predicting the minority class in all cases. The complete example is listed below.

```
# evaluate undersampling with logistic regression on the imbalanced german credit dataset
from numpy import mean
from numpy import std
from pandas import read_csv
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import MinMaxScaler
from sklearn.compose import ColumnTransformer
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.metrics import fbeta_score
from sklearn.metrics import make_scorer
from matplotlib import pyplot
from sklearn.linear_model import LogisticRegression
from imblearn.pipeline import Pipeline
from imblearn.under_sampling import TomekLinks
from imblearn.under_sampling import EditedNearestNeighbours
from imblearn.under_sampling import RepeatedEditedNearestNeighbours
from imblearn.under_sampling import NeighbourhoodCleaningRule
from imblearn.under_sampling import OneSidedSelection

# load the dataset
def load_dataset(full_path):
    # load the dataset as a numpy array
    datafram = read_csv(full_path, header=None)
    # split into inputs and outputs
    last_ix = len(datafram.columns) - 1
    X, y = datafram.drop(last_ix, axis=1), datafram[last_ix]
    # select categorical and numerical features
    cat_ix = X.select_dtypes(include=['object', 'bool']).columns
    num_ix = X.select_dtypes(include=['int64', 'float64']).columns
    # label encode the target variable to have the classes 0 and 1
    y = LabelEncoder().fit_transform(y)
    return X.values, y, cat_ix, num_ix

# calculate f2-measure
def f2_measure(y_true, y_pred):
    return fbeta_score(y_true, y_pred, beta=2)

# evaluate a model
def evaluate_model(X, y, model):
    # define evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # define the model evaluation metric
    metric = make_scorer(f2_measure)
    # evaluate model
    scores = cross_val_score(model, X, y, scoring=metric, cv=cv, n_jobs=-1)
    return scores

# define undersampling models to test
def get_models():
    models, names = list(), list()
    # TL
    models.append(TomekLinks())
    names.append('TL')
```

```

# ENN
models.append(EditedNearestNeighbours())
names.append('ENN')
# RENN
models.append(RepeatedEditedNearestNeighbours())
names.append('RENN')
# OSS
models.append(OneSidedSelection())
names.append('OSS')
# NCR
models.append(NeighbourhoodCleaningRule())
names.append('NCR')
return models, names

# define the location of the dataset
full_path = 'german.csv'
# load the dataset
X, y, cat_ix, num_ix = load_dataset(full_path)
# define models
models, names = get_models()
results = list()
# evaluate each model
for i in range(len(models)):
    # define model to evaluate
    model = LogisticRegression(solver='liblinear', class_weight='balanced')
    # one hot encode categorical, normalize numerical
    ct = ColumnTransformer([('c', OneHotEncoder(), cat_ix), ('n', MinMaxScaler(), num_ix)])
    # scale, then undersample, then fit model
    pipeline = Pipeline(steps=[('t', ct), ('s', models[i]), ('m', model)])
    # evaluate the model and store results
    scores = evaluate_model(X, y, pipeline)
    results.append(scores)
    # summarize and store
    print('>%s %.3f (%.3f)' % (names[i], mean(scores), std(scores)))
# plot the results
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()

```

Listing 28.29: Calculate the performance of data sampling algorithms for the German dataset.

Running the example evaluates the logistic regression algorithm with five different undersampling techniques.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that three of the five undersampling techniques resulted in an F2-measure that provides an improvement over the baseline of 0.682. Specifically, ENN, RENN and NCR, with repeated edited neighbors (RENN) resulting in the best performance with an F2-measure of about 0.716.

```

>TL 0.669 (0.057)
>ENN 0.706 (0.048)
>RENN 0.714 (0.041)
>OSS 0.670 (0.054)

```

```
>NCR 0.693 (0.052)
```

Listing 28.30: Example output from calculating the performance of machine learning models on the German dataset.

Box and whisker plots are created for each evaluated undersampling technique, showing that they generally have the same spread. It is encouraging to see that for the well performing methods, the boxes spread up around 0.8, and the mean and median for all three methods are around 0.7. This highlights that the distributions are skewing high and are let down on occasion by a few bad evaluations.

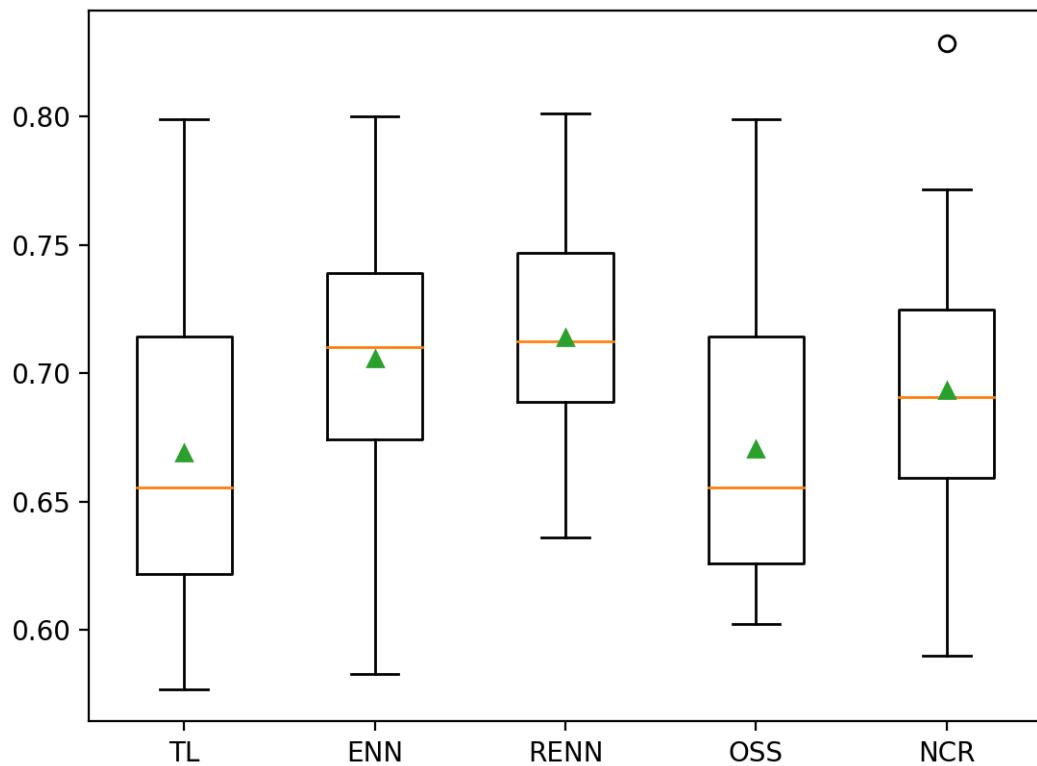


Figure 28.3: Box and Whisker Plot of Logistic Regression With Undersampling on the Imbalanced German Credit Dataset.

Next, let's see how we might use a final model to make predictions on new data.

28.6 Make Prediction on New Data

Given the variance in results, a selection of any of the undersampling methods is probably sufficient. In this case, we will select logistic regression with Repeated ENN. This model had an F2-measure of about 0.716 on our test harness. We will use this as our final model and use it to make predictions on new data. First, we can define the model as a pipeline.

```

...
# define model to evaluate
model = LogisticRegression(solver='liblinear', class_weight='balanced')
# one hot encode categorical, normalize numerical
ct = ColumnTransformer([('c',OneHotEncoder(),cat_ix), ('n',MinMaxScaler(),num_ix)])
# scale, then oversample, then fit model
pipeline = Pipeline(steps=[('t',ct), ('s', RepeatedEditedNearestNeighbours()), ('m',model)])

```

Listing 28.31: Example of defining the final model.

Once defined, we can fit it on the entire training dataset.

```

...
# fit the model
pipeline.fit(X, y)

```

Listing 28.32: Example of fitting the final model.

Once fit, we can use it to make predictions for new data by calling the `predict()` function. This will return the class label of 0 for *good customer*, or 1 for *bad customer*. Importantly, we must use the `ColumnTransformer` that was fit on the training dataset in the `Pipeline` to correctly prepare new data using the same transforms. For example:

```

...
# define a row of data
row = [...]
# make prediction
yhat = pipeline.predict([row])

```

Listing 28.33: Example of making a prediction with the fit final model.

To demonstrate this, we can use the fit model to make some predictions of labels for a few cases where we know if the case is a good customer or bad. The complete example is listed below.

```

# fit a model and make predictions for the german credit dataset
from pandas import read_csv
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import MinMaxScaler
from sklearn.compose import ColumnTransformer
from sklearn.linear_model import LogisticRegression
from imblearn.pipeline import Pipeline
from imblearn.under_sampling import RepeatedEditedNearestNeighbours

# load the dataset
def load_dataset(full_path):
    # load the dataset as a numpy array
    dataframe = read_csv(full_path, header=None)
    # split into inputs and outputs
    last_ix = len(dataframe.columns) - 1
    X, y = dataframe.drop(last_ix, axis=1), dataframe[last_ix]
    # select categorical and numerical features
    cat_ix = X.select_dtypes(include=['object', 'bool']).columns
    num_ix = X.select_dtypes(include=['int64', 'float64']).columns
    # label encode the target variable to have the classes 0 and 1
    y = LabelEncoder().fit_transform(y)

```

```

    return X.values, y, cat_ix, num_ix

# define the location of the dataset
full_path = 'german.csv'
# load the dataset
X, y, cat_ix, num_ix = load_dataset(full_path)
# define model to evaluate
model = LogisticRegression(solver='liblinear', class_weight='balanced')
# one hot encode categorical, normalize numerical
ct = ColumnTransformer([('c',OneHotEncoder(),cat_ix), ('n',MinMaxScaler(),num_ix)])
# scale, then undersample, then fit model
pipeline = Pipeline(steps=[('t',ct), ('s', RepeatedEditedNearestNeighbours()), ('m',model)])
# fit the model
pipeline.fit(X, y)
# evaluate on some good customers cases (known class 0)
print('Good Customers:')
data = [['A11', 6, 'A34', 'A43', 1169, 'A65', 'A75', 4, 'A93', 'A101', 4, 'A121', 67,
         'A143', 'A152', 2, 'A173', 1, 'A192', 'A201'],
        ['A14', 12, 'A34', 'A46', 2096, 'A61', 'A74', 2, 'A93', 'A101', 3, 'A121', 49, 'A143',
         'A152', 1, 'A172', 2, 'A191', 'A201'],
        ['A11', 42, 'A32', 'A42', 7882, 'A61', 'A74', 2, 'A93', 'A103', 4, 'A122', 45, 'A143',
         'A153', 1, 'A173', 2, 'A191', 'A201']]
for row in data:
    # make prediction
    yhat = pipeline.predict([row])
    # get the label
    label = yhat[0]
    # summarize
    print('>Predicted=%d (expected 0)' % (label))
# evaluate on some bad customers (known class 1)
print('Bad Customers:')
data = [['A13', 18, 'A32', 'A43', 2100, 'A61', 'A73', 4, 'A93', 'A102', 2, 'A121', 37,
         'A142', 'A152', 1, 'A173', 1, 'A191', 'A201'],
        ['A11', 24, 'A33', 'A40', 4870, 'A61', 'A73', 3, 'A93', 'A101', 4, 'A124', 53, 'A143',
         'A153', 2, 'A173', 2, 'A191', 'A201'],
        ['A11', 24, 'A32', 'A43', 1282, 'A62', 'A73', 4, 'A92', 'A101', 2, 'A123', 32, 'A143',
         'A152', 1, 'A172', 1, 'A191', 'A201']]
for row in data:
    # make prediction
    yhat = pipeline.predict([row])
    # get the label
    label = yhat[0]
    # summarize
    print('>Predicted=%d (expected 1)' % (label))

```

Listing 28.34: Fit a model and use it to make predictions.

Running the example first fits the model on the entire training dataset. Then the fit model used to predict the label of a good customer for cases chosen from the dataset file. We can see that most cases are correctly predicted. This highlights that although we chose a good model, it is not perfect. Then some cases of actual bad customers are used as input to the model and the label is predicted. As we might have hoped, the correct labels are predicted for all cases.

```

Good Customers:
>Predicted=0 (expected 0)
>Predicted=0 (expected 0)

```

```
>Predicted=1 (expected 0)
Bad Customers:
>Predicted=1 (expected 1)
>Predicted=1 (expected 1)
>Predicted=1 (expected 1)
```

Listing 28.35: Example output from fitting a model and use it to make predictions.

28.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

28.7.1 Books

- *Machine Learning, Neural and Statistical Classification*, 1994.
<https://amzn.to/33oQT1q>

28.7.2 APIs

- pandas.DataFrame.select_dtypes API.
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.select_dtypes.html
- sklearn.metrics.fbeta_score API.
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.fbeta_score.html
- sklearn.compose.ColumnTransformer API.
<https://scikit-learn.org/stable/modules/generated/sklearn.compose.ColumnTransformer.html>
- sklearn.preprocessing.OneHotEncoder API.
<https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html>
- imblearn.pipeline.Pipeline API.
<https://imbalanced-learn.org/stable/generated/imblearn.pipeline.Pipeline.html>

28.7.3 Dataset

- Statlog (German Credit Data) Dataset, UCI Machine Learning Repository.
[https://archive.ics.uci.edu/ml/datasets/Statlog+\(German+Credit+Data\)](https://archive.ics.uci.edu/ml/datasets/Statlog+(German+Credit+Data))
- German Credit Dataset.
<https://raw.githubusercontent.com/jbrownlee/Datasets/master/german.csv>
- German Credit Dataset Description
<https://raw.githubusercontent.com/jbrownlee/Datasets/master/german.names>

28.8 Summary

In this tutorial, you discovered how to develop and evaluate a model for the imbalanced German credit classification dataset. Specifically, you learned:

- How to load and explore the dataset and generate ideas for data preparation and model selection.
- How to evaluate a suite of machine learning models and improve their performance with undersampling techniques.
- How to fit a final model and use it to predict class labels for specific cases.

28.8.1 Next

In the next tutorial, you will discover how to systematically work through the Mammography imbalanced classification dataset.

Chapter 29

Project: Microcalcification Classification

Cancer detection is a popular example of an imbalanced classification problem because there are often significantly more cases of non-cancer than actual cancer. A standard imbalanced classification dataset is the mammography dataset that involves detecting breast cancer from radiological scans, specifically the presence of clusters of microcalcifications that appear bright on a mammogram. This dataset was constructed by scanning the images, segmenting them into candidate objects, and using computer vision techniques to describe each candidate object.

It is a popular dataset for imbalanced classification because of the severe class imbalance, specifically where 98 percent of candidate microcalcifications are not cancer and only 2 percent were labeled as cancer by an experienced radiographer. In this tutorial, you will discover how to develop and evaluate models for the imbalanced mammography cancer classification dataset. After completing this tutorial, you will know:

- How to load and explore the dataset and generate ideas for data preparation and model selection.
- How to evaluate a suite of machine learning models and improve their performance with data cost-sensitive techniques.
- How to fit a final model and use it to predict class labels for specific cases.

Let's get started.

29.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Mammography Dataset
2. Explore the Dataset
3. Model Test and Baseline Result
4. Evaluate Models
5. Make Predictions on New Data

29.2 Mammography Dataset

In this project, we will use a standard imbalanced machine learning dataset referred to as the *mammography* dataset or sometimes *Woods Mammography*. The dataset is credited to Kevin Woods, et al. and the 1993 paper titled *Comparative Evaluation Of Pattern Recognition Techniques For Detection Of Microcalcifications In Mammography*. The focus of the problem is on detecting breast cancer from radiological scans, specifically the presence of clusters of microcalcifications that appear bright on a mammogram.

The dataset involved first started with 24 mammograms with a known cancer diagnosis that were scanned. The images were then pre-processed using image segmentation computer vision algorithms to extract candidate objects from the mammogram images. Once segmented, the objects were then manually labeled by an experienced radiologist. A total of 29 features were extracted from the segmented objects thought to be most relevant to pattern recognition, which was reduced to 18, then finally to six, as follows (taken directly from the paper):

- Area of object (in pixels).
- Average gray level of the object.
- Gradient strength of the object's perimeter pixels.
- Root mean square noise fluctuation in the object.
- Contrast, average gray level of the object minus the average of a two-pixel wide border surrounding the object.
- A low order moment based on shape descriptor.

There are two classes and the goal is to distinguish between microcalcifications and non-microcalcifications using the features for a given segmented object.

- **Non-microcalcifications:** negative case, or majority class.
- **Microcalcifications:** positive case, or minority class.

A number of models were evaluated and compared in the original paper, such as neural networks, decision trees, and k -nearest neighbors. Models were evaluated using ROC Curves and compared using the area under ROC Curve, or ROC AUC for short.

ROC Curves and area under ROC Curves were chosen with the intent to minimize the false-positive rate (complement of the specificity) and maximize the true-positive rate (sensitivity), the two axes of the ROC Curve. The use of the ROC Curves also suggests the desire for a probabilistic model from which an operator can select a probability threshold as the cut-off between the acceptable false positive and true positive rates. Their results suggested a *linear classifier* (seemingly a Gaussian Naive Bayes classifier) performed the best with a ROC AUC of 0.936 averaged over 100 runs. Next, let's take a closer look at the data.

29.3 Explore the Dataset

The Mammography dataset is a widely used standard machine learning dataset, used to explore and demonstrate many techniques designed specifically for imbalanced classification. One example is the popular SMOTE data oversampling technique. A version of this dataset was made available that has some differences from the dataset described in the original paper. First, download the dataset and save it in your current working directory with the name `mammography.csv`.

- Download Mammography Dataset (`mammography.csv`). ¹

Review the contents of the file. The first few lines of the file should look as follows:

```
0.23001961,5.0725783,-0.27606055,0.83244412,-0.37786573,0.4803223,'-1'  
0.15549112,-0.16939038,0.67065219,-0.85955255,-0.37786573,-0.94572324,'-1'  
-0.78441482,-0.44365372,5.6747053,-0.85955255,-0.37786573,-0.94572324,'-1'  
0.54608818,0.13141457,-0.45638679,-0.85955255,-0.37786573,-0.94572324,'-1'  
-0.10298725,-0.3949941,-0.14081588,0.97970269,-0.37786573,1.0135658,'-1'  
...
```

Listing 29.1: Example of rows of data from the mammography dataset.

We can see that the dataset has six rather than the seven input variables. It is possible that the first input variable listed in the paper (area in pixels) was removed from this version of the dataset. The input variables are numerical (real-valued) and the target variable is the string with ‘-1’ for the majority class and ‘1’ for the minority class. These values will need to be encoded as 0 and 1 respectively to meet the expectations of classification algorithms on binary imbalanced classification problems. The dataset can be loaded as a `DataFrame` using the `read_csv()` Pandas function, specifying the location and the fact that there is no header line.

```
...  
# define the dataset location  
filename = 'mammography.csv'  
# load the csv file as a data frame  
dataframe = read_csv(filename, header=None)
```

Listing 29.2: Example of loading the mammography dataset.

Once loaded, we can summarize the number of rows and columns by printing the shape of the `DataFrame`.

```
...  
# summarize the shape of the dataset  
print(dataframe.shape)
```

Listing 29.3: Example of summarizing the shape of the loaded dataset.

We can also summarize the number of examples in each class using the `Counter` object.

```
...  
# summarize the class distribution  
target = dataframe.values[:, -1]  
counter = Counter(target)  
for k, v in counter.items():
```

¹<https://raw.githubusercontent.com/jbrownlee/Datasets/master/mammography.csv>

```

per = v / len(target) * 100
print('Class=%s, Count=%d, Percentage=%.3f%%' % (k, v, per))

```

Listing 29.4: Example of summarizing the class distribution for the loaded dataset.

Tying this together, the complete example of loading and summarizing the dataset is listed below.

```

# load and summarize the dataset
from pandas import read_csv
from collections import Counter
# define the dataset location
filename = 'mammography.csv'
# load the csv file as a data frame
dataframe = read_csv(filename, header=None)
# summarize the shape of the dataset
print(dataframe.shape)
# summarize the class distribution
target = dataframe.values[:, -1]
counter = Counter(target)
for k,v in counter.items():
    per = v / len(target) * 100
    print('Class=%s, Count=%d, Percentage=%.3f%%' % (k, v, per))

```

Listing 29.5: Load and summarize the mammography dataset.

Running the example first loads the dataset and confirms the number of rows and columns, that is 11,183 rows and six input variables and one target variable. The class distribution is then summarized, confirming the severe class imbalance with approximately 98 percent for the majority class (no cancer) and approximately 2 percent for the minority class (cancer).

```

(11183, 7)
Class='-1', Count=10923, Percentage=97.675%
Class='1', Count=260, Percentage=2.325%

```

Listing 29.6: Example output from loading and summarizing the mammography dataset.

The dataset appears to generally match the dataset described in the SMOTE paper. Specifically in terms of the ratio of negative to positive examples.

A typical mammography dataset might contain 98% normal pixels and 2% abnormal pixels.

— SMOTE: *Synthetic Minority Over-sampling Technique*, 2002.

Also, the specific number of examples in the minority and majority classes also matches the paper.

The experiments were conducted on the mammography dataset. There were 10923 examples in the majority class and 260 examples in the minority class originally.

— SMOTE: *Synthetic Minority Over-sampling Technique*, 2002.

I believe this is the same dataset, although I cannot explain the mismatch in the number of input features, e.g. six compared to seven in the original paper. We can also take a look at the distribution of the six numerical input variables by creating a histogram for each. The complete example is listed below.

```
# create histograms of numeric input variables
from pandas import read_csv
from matplotlib import pyplot
# define the dataset location
filename = 'mammography.csv'
# load the csv file as a data frame
df = read_csv(filename, header=None)
# histograms of all variables
df.hist()
pyplot.show()
```

Listing 29.7: Plot histograms of the mammography dataset.

Running the example creates the figure with one histogram subplot for each of the six numerical input variables in the dataset. We can see that the variables have differing scales and that most of the variables have an exponential distribution, e.g. most cases falling into one bin, and the rest falling into a long tail. The final variable appears to have a bimodal distribution.

Depending on the choice of modeling algorithms, we would expect scaling the distributions to the same range to be useful, and perhaps the use of some power transforms.

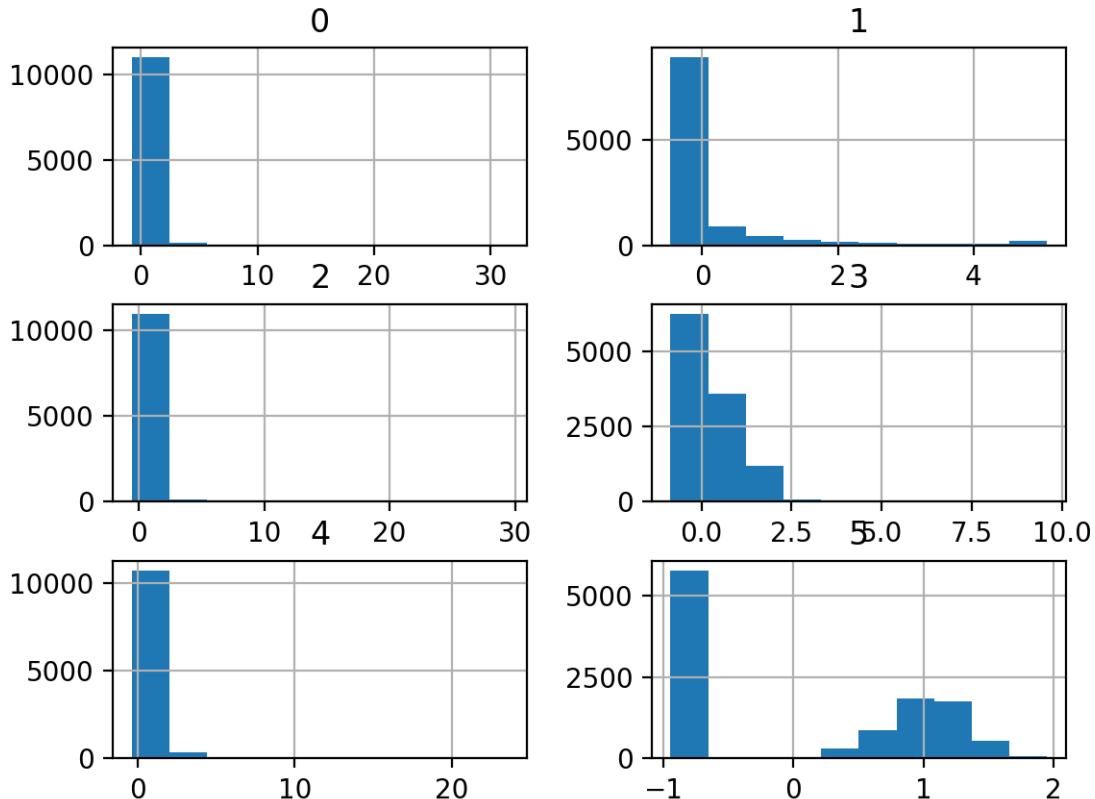


Figure 29.1: Histogram Plots of the Numerical Input Variables for the Mammography Dataset.

We can also create a scatter plot for each pair of input variables, called a scatter plot matrix. This can be helpful to see if any variables relate to each other or change in the same direction,

e.g. are correlated. We can also color the dots of each scatter plot according to the class label. In this case, the majority class (no cancer) will be mapped to blue dots and the minority class (cancer) will be mapped to red dots. The complete example is listed below.

```
# create pairwise scatter plots of numeric input variables
from pandas import read_csv
from pandas.plotting import scatter_matrix
from matplotlib import pyplot
# define the dataset location
filename = 'mammography.csv'
# load the csv file as a data frame
df = read_csv(filename, header=None)
# define a mapping of class values to colors
color_dict = {"-1":'blue', "1":'red'}
# map each row to a color based on the class value
colors = [color_dict[str(x)] for x in df.values[:, -1]]
# pairwise scatter plots of all numerical variables
scatter_matrix(df, diagonal='kde', color=colors)
pyplot.show()
```

Listing 29.8: Scatter plot matrix of the mammography dataset.

Running the example creates a figure showing the scatter plot matrix, with six plots by six plots, comparing each of the six numerical input variables with each other. The diagonal of the matrix shows the density distribution of each variable. Each pairing appears twice both above and below the top-left to bottom-right diagonal, providing two ways to review the same variable interactions. We can see that the distributions for many variables do differ for the two-class labels, suggesting that some reasonable discrimination between the cancer and no cancer cases will be feasible.

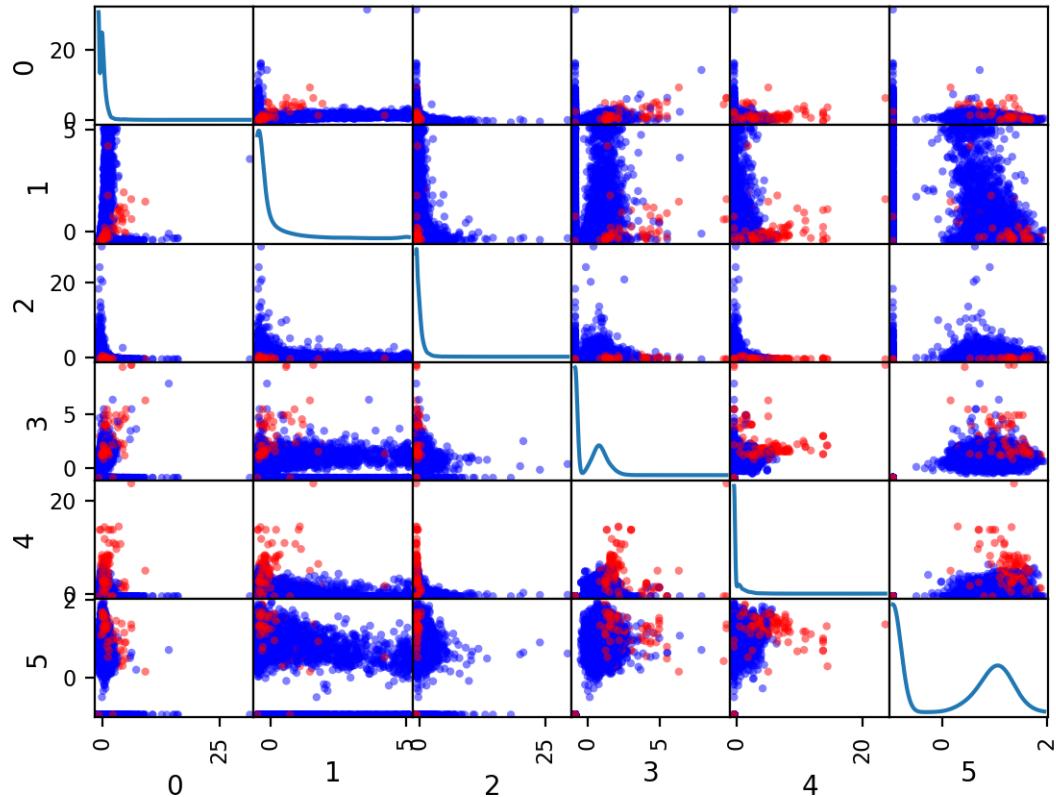


Figure 29.2: Scatter Plot Matrix by Class for the Numerical Input Variables in the Mammography Dataset.

Now that we have reviewed the dataset, let's look at developing a test harness for evaluating candidate models.

29.4 Model Test and Baseline Result

We will evaluate candidate models using repeated stratified k -fold cross-validation. The k -fold cross-validation procedure provides a good general estimate of model performance that is not too optimistically biased, at least compared to a single train-test split. We will use $k = 10$, meaning each fold will contain about $\frac{11183}{10}$ or about 1,118 examples. Stratified means that each fold will contain the same mixture of examples by class, that is about 98 percent to 2 percent no-cancer to cancer examples. Repetition indicates that the evaluation process will be performed multiple times to help avoid fluke results and better capture the variance of the chosen model. We will use three repeats.

This means a single model will be fit and evaluated 10×3 (30) times and the mean and standard deviation of these runs will be reported. This can be achieved using the `RepeatedStratifiedKFold` scikit-learn class. We will evaluate and compare models using the area under ROC Curve or ROC AUC calculated via the `roc_auc_score()` function (for more on the ROC AUC, see Chapter 7). We can define a function to load the dataset and split

the columns into input and output variables. We will correctly encode the class labels as 0 and 1. The `load_dataset()` function below implements this.

```
# load the dataset
def load_dataset(full_path):
    # load the dataset as a numpy array
    data = read_csv(full_path, header=None)
    # retrieve numpy array
    data = data.values
    # split into input and output elements
    X, y = data[:, :-1], data[:, -1]
    # label encode the target variable to have the classes 0 and 1
    y = LabelEncoder().fit_transform(y)
    return X, y
```

Listing 29.9: Example of a function for loading and preparing the dataset for modeling.

We can then define a function that will evaluate a given model on the dataset and return a list of ROC AUC scores for each fold and repeat. The `evaluate_model()` function below implements this, taking the dataset and model as arguments and returning the list of scores.

```
# evaluate a model
def evaluate_model(X, y, model):
    # define evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # evaluate model
    scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
    return scores
```

Listing 29.10: Example of a function for evaluating a defined model.

Finally, we can evaluate a baseline model on the dataset using this test harness. A model that predicts a random class label in proportion to the base rate of each class will result in a ROC AUC of 0.5, the baseline in performance on this dataset. This is a so-called *no skill* classifier. This can be achieved using the `DummyClassifier` class from the scikit-learn library and setting the `strategy` argument to ‘stratified’.

```
...
# define the reference model
model = DummyClassifier(strategy='stratified')
```

Listing 29.11: Example of defining the baseline model.

Once the model is evaluated, we can report the mean and standard deviation of the ROC AUC scores directly.

```
...
# evaluate the model
scores = evaluate_model(X, y, model)
# summarize performance
print('Mean ROC AUC: %.3f (%.3f)' % (mean(scores), std(scores)))
```

Listing 29.12: Example of evaluating the model and summarizing the performance.

Tying this together, the complete example of loading the dataset, evaluating a baseline model, and reporting the performance is listed below.

```

# test harness and baseline model evaluation
from collections import Counter
from numpy import mean
from numpy import std
from pandas import read_csv
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.dummy import DummyClassifier

# load the dataset
def load_dataset(full_path):
    # load the dataset as a numpy array
    data = read_csv(full_path, header=None)
    # retrieve numpy array
    data = data.values
    # split into input and output elements
    X, y = data[:, :-1], data[:, -1]
    # label encode the target variable to have the classes 0 and 1
    y = LabelEncoder().fit_transform(y)
    return X, y

# evaluate a model
def evaluate_model(X, y, model):
    # define evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # evaluate model
    scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
    return scores

# define the location of the dataset
full_path = 'mammography.csv'
# load the dataset
X, y = load_dataset(full_path)
# summarize the loaded dataset
print(X.shape, y.shape, Counter(y))
# define the reference model
model = DummyClassifier(strategy='stratified')
# evaluate the model
scores = evaluate_model(X, y, model)
# summarize performance
print('Mean ROC AUC: %.3f (%.3f)' % (mean(scores), std(scores)))

```

Listing 29.13: Calculate the baseline performance for the mammography dataset.

Running the example first loads and summarizes the dataset. We can see that we have the correct number of rows loaded, and that we have six computer vision derived input variables. Importantly, we can see that the class labels have the correct mapping to integers with 0 for the majority class and 1 for the minority class, customary for imbalanced binary classification datasets. Next, the average of the ROC AUC scores is reported. As expected, the no-skill classifier achieves the worst-case performance of a mean ROC AUC of approximately 0.5. This provides a baseline in performance, above which a model can be considered skillful on this dataset.

(11183, 6) (11183,) Counter({0: 10923, 1: 260})

```
Mean ROC AUC: 0.503 (0.016)
```

Listing 29.14: Example output from calculating the baseline performance for the mammography dataset.

Now that we have a test harness and a baseline in performance, we can begin to evaluate some models on this dataset.

29.5 Evaluate Models

In this section, we will evaluate a suite of different techniques on the dataset using the test harness developed in the previous section. The goal is to both demonstrate how to work through the problem systematically and to demonstrate the capability of some techniques designed for imbalanced classification problems. The reported performance is good, but not highly optimized (e.g. hyperparameters are not tuned).

29.5.1 Evaluate Machine Learning Algorithms

Let's start by evaluating a mixture of machine learning models on the dataset. It can be a good idea to spot-check a suite of different linear and nonlinear algorithms on a dataset to quickly flush out what works well and deserves further attention, and what doesn't. We will evaluate the following machine learning models on the mammography dataset:

- Logistic Regression (LR)
- Support Vector Machine (SVM)
- Bagged Decision Trees (BAG)
- Random Forest (RF)
- Gradient Boosting Machine (GBM)

We will use mostly default model hyperparameters, with the exception of the number of trees in the ensemble algorithms, which we will set to a reasonable default of 1,000. We will define each model in turn and add them to a list so that we can evaluate them sequentially. The `get_models()` function below defines the list of models for evaluation, as well as a list of model short names for plotting the results later.

```
# define models to test
def get_models():
    models, names = list(), list()
    # LR
    models.append(LogisticRegression(solver='lbfgs'))
    names.append('LR')
    # SVM
    models.append(SVC(gamma='scale'))
    names.append('SVM')
    # Bagging
    models.append(BaggingClassifier(n_estimators=1000))
    names.append('BAG')
```

```

# RF
models.append(RandomForestClassifier(n_estimators=1000))
names.append('RF')
# GBM
models.append(GradientBoostingClassifier(n_estimators=1000))
names.append('GBM')
return models, names

```

Listing 29.15: Example of a function for defining the models to evaluate.

We can then enumerate the list of models in turn and evaluate each, reporting the mean ROC AUC and storing the scores for later plotting.

```

...
# define models
models, names = get_models()
results = list()
# evaluate each model
for i in range(len(models)):
    # evaluate the model and store results
    scores = evaluate_model(X, y, models[i])
    results.append(scores)
    # summarize and store
    print('>%s %.3f (%.3f)' % (names[i], mean(scores), std(scores)))

```

Listing 29.16: Example of evaluating the defined models.

At the end of the run, we can plot each sample of scores as a box and whisker plot with the same scale so that we can directly compare the distributions.

```

...
# plot the results
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()

```

Listing 29.17: Example of summarizing performance using box and whisker plots.

Tying this all together, the complete example of evaluating a suite of machine learning algorithms on the mammography dataset is listed below.

```

# spot check machine learning algorithms on the mammography dataset
from numpy import mean
from numpy import std
from pandas import read_csv
from matplotlib import pyplot
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import BaggingClassifier

# load the dataset
def load_dataset(full_path):
    # load the dataset as a numpy array
    data = read_csv(full_path, header=None)

```

```

# retrieve numpy array
data = data.values
# split into input and output elements
X, y = data[:, :-1], data[:, -1]
# label encode the target variable to have the classes 0 and 1
y = LabelEncoder().fit_transform(y)
return X, y

# evaluate a model
def evaluate_model(X, y, model):
    # define evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # evaluate model
    scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
    return scores

# define models to test
def get_models():
    models, names = list(), list()
    # LR
    models.append(LogisticRegression(solver='lbfgs'))
    names.append('LR')
    # SVM
    models.append(SVC(gamma='scale'))
    names.append('SVM')
    # Bagging
    models.append(BaggingClassifier(n_estimators=1000))
    names.append('BAG')
    # RF
    models.append(RandomForestClassifier(n_estimators=1000))
    names.append('RF')
    # GBM
    models.append(GradientBoostingClassifier(n_estimators=1000))
    names.append('GBM')
    return models, names

# define the location of the dataset
full_path = 'mammography.csv'
# load the dataset
X, y = load_dataset(full_path)
# define models
models, names = get_models()
results = list()
# evaluate each model
for i in range(len(models)):
    # evaluate the model and store results
    scores = evaluate_model(X, y, models[i])
    results.append(scores)
    # summarize and store
    print('>%s %.3f (%.3f)' % (names[i], mean(scores), std(scores)))
# plot the results
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()

```

Listing 29.18: Calculate the performance of machine learning models on the mammography dataset.

Running the example evaluates each algorithm in turn and reports the mean and standard deviation ROC AUC.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that all of the tested algorithms have skill, achieving a ROC AUC above the naive model with 0.5. The results suggest that the ensemble of decision tree algorithms performs better on this dataset with perhaps Random Forest performing the best, with a ROC AUC of about 0.950. It is interesting to note that this is better than the ROC AUC described in the paper of 0.93, although we used a different model evaluation procedure. The evaluation was a little unfair to the LR and SVM algorithms as we did not scale the input variables prior to fitting the model.

```
>LR 0.919 (0.040)
>SVM 0.880 (0.049)
>BAG 0.941 (0.041)
>RF 0.950 (0.036)
>GBM 0.918 (0.037)
```

Listing 29.19: Example output from calculating the performance of machine learning models on the mammography dataset.

A figure is created showing one box and whisker plot for each algorithm's sample of results. The box shows the middle 50 percent of the data, the orange line in the middle of each box shows the median of the sample, and the green triangle in each box shows the mean of the sample. We can see that both BAG and RF have a tight distribution and a mean and median that closely align, perhaps suggesting a non-skewed and Gaussian distribution of scores, i.e. stable.

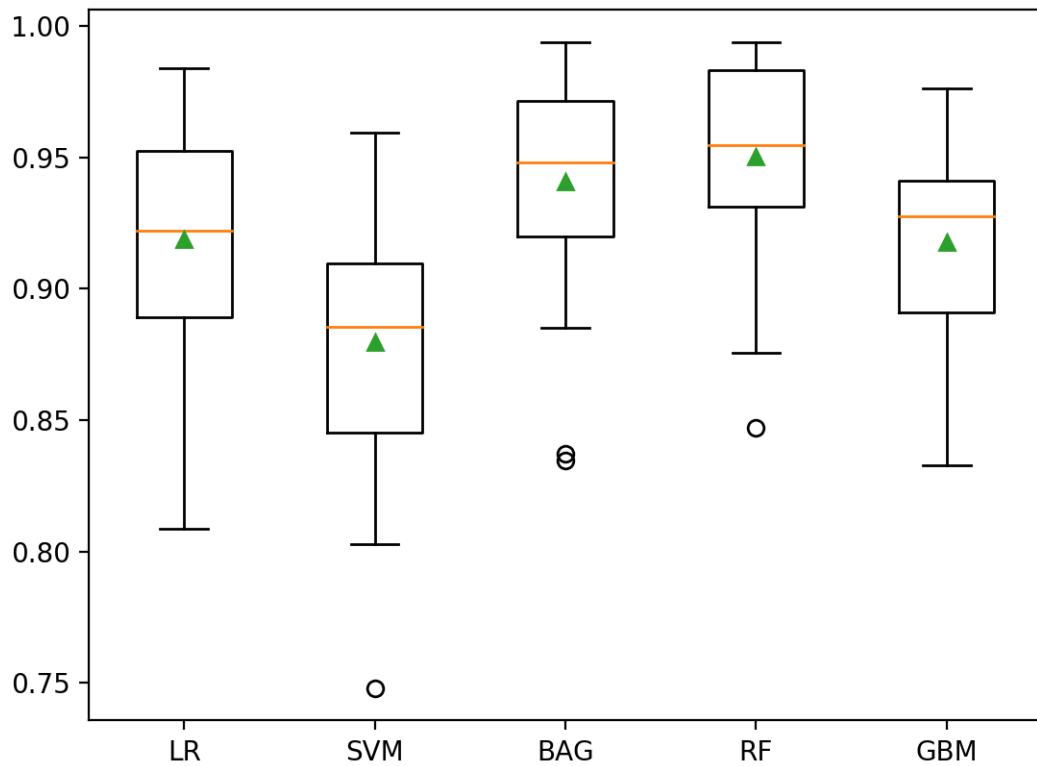


Figure 29.3: Box and Whisker Plot of Machine Learning Models on the Imbalanced Mammography Dataset.

Now that we have a good first set of results, let's see if we can improve them with cost-sensitive classifiers.

29.5.2 Evaluate Cost-Sensitive Algorithms

Some machine learning algorithms can be adapted to pay more attention to one class than another when fitting the model. These are referred to as cost-sensitive machine learning models and they can be used for imbalanced classification by specifying a cost that is inversely proportional to the class distribution. For example, with a 98 percent to 2 percent distribution for the majority and minority classes, we can specify to give errors on the minority class a weighting of 98 and errors for the majority class a weighting of 2.

Three algorithms that offer this capability are:

- Logistic Regression (LR)
- Support Vector Machine (SVM)
- Random Forest (RF)

This can be achieved in scikit-learn by setting the `class_weight` argument to ‘`balanced`’ to make these algorithms cost-sensitive. For example, the updated `get_models()` function below defines the cost-sensitive version of these three algorithms to be evaluated on our dataset.

```
# define models to test
def get_models():
    models, names = list(), list()
    # LR
    models.append(LogisticRegression(solver='lbfgs', class_weight='balanced'))
    names.append('LR')
    # SVM
    models.append(SVC(gamma='scale', class_weight='balanced'))
    names.append('SVM')
    # RF
    models.append(RandomForestClassifier(n_estimators=1000))
    names.append('RF')
    return models, names
```

Listing 29.20: Example of a function for defining the cost-sensitive models to evaluate.

Additionally, when exploring the dataset, we noticed that many of the variables had a seemingly exponential data distribution. Sometimes we can better spread the data for a variable by using a power transform on each variable. This will be particularly helpful to the LR and SVM algorithms and may also help the RF algorithm. We can implement this within each fold of the cross-validation model evaluation process using a `Pipeline`. The first step will fit the `PowerTransformer` on the training set folds and apply it to the training and test set folds. The second step will be the model that we are evaluating. The pipeline can then be evaluated directly using our `evaluate_model()` function, for example:

```
...
# defines pipeline steps
steps = [('p', PowerTransformer()), ('m', models[i])]
# define pipeline
pipeline = Pipeline(steps=steps)
# evaluate the pipeline and store results
scores = evaluate_model(X, y, pipeline)
```

Listing 29.21: Example of defining a pipeline for a cost-sensitive model.

Tying this together, the complete example of evaluating power transformed cost-sensitive machine learning algorithms on the mammography dataset is listed below.

```
# cost-sensitive machine learning algorithms on the mammography dataset
from numpy import mean
from numpy import std
from pandas import read_csv
from matplotlib import pyplot
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import PowerTransformer
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
```

```
# load the dataset
def load_dataset(full_path):
    # load the dataset as a numpy array
    data = read_csv(full_path, header=None)
    # retrieve numpy array
    data = data.values
    # split into input and output elements
    X, y = data[:, :-1], data[:, -1]
    # label encode the target variable to have the classes 0 and 1
    y = LabelEncoder().fit_transform(y)
    return X, y

# evaluate a model
def evaluate_model(X, y, model):
    # define evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # evaluate model
    scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
    return scores

# define models to test
def get_models():
    models, names = list(), list()
    # LR
    models.append(LogisticRegression(solver='lbfgs', class_weight='balanced'))
    names.append('LR')
    # SVM
    models.append(SVC(gamma='scale', class_weight='balanced'))
    names.append('SVM')
    # RF
    models.append(RandomForestClassifier(n_estimators=1000))
    names.append('RF')
    return models, names

# define the location of the dataset
full_path = 'mammography.csv'
# load the dataset
X, y = load_dataset(full_path)
# define models
models, names = get_models()
results = list()
# evaluate each model
for i in range(len(models)):
    # defines pipeline steps
    steps = [('p', PowerTransformer()), ('m', models[i])]
    # define pipeline
    pipeline = Pipeline(steps=steps)
    # evaluate the pipeline and store results
    scores = evaluate_model(X, y, pipeline)
    results.append(scores)
    # summarize and store
    print('>%s %.3f (%.3f)' % (names[i], mean(scores), std(scores)))
# plot the results
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 29.22: Calculate the performance of cost-sensitive models on the mammography dataset.

Running the example evaluates each algorithm in turn and reports the mean and standard deviation ROC AUC.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that all three of the tested algorithms achieved a lift on ROC AUC compared to their non-transformed and cost-insensitive versions. It would be interesting to repeat the experiment without the transform to see if it was the transform or the cost-sensitive version of the algorithms, or both that resulted in the lifts in performance.

In this case, we can see the SVM achieved the best performance, performing better than RF in this and the previous section and achieving a mean ROC AUC of about 0.957.

```
>LR 0.922 (0.036)
>SVM 0.957 (0.024)
>RF 0.951 (0.035)
```

Listing 29.23: Example output from calculating the performance of cost-sensitive models on the mammography dataset.

Box and whisker plots are then created comparing the distribution of ROC AUC scores. The SVM distribution appears compact compared to the other two models. As such the performance is likely stable and may make a good choice for a final model.

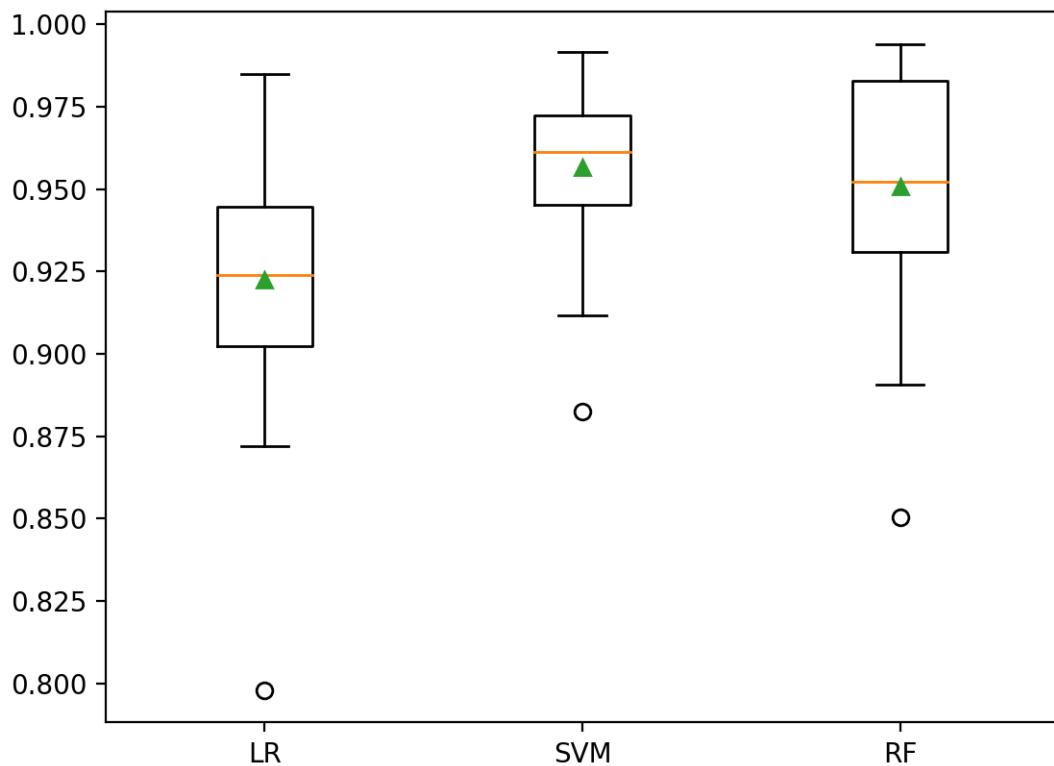


Figure 29.4: Box and Whisker Plots of Cost-Sensitive Machine Learning Models on the Imbalanced Mammography Dataset.

Next, let's see how we might use a final model to make predictions on new data.

29.6 Make Predictions on New Data

In this section, we will fit a final model and use it to make predictions on single rows of data. We will use the cost-sensitive version of the SVM model as the final model and a power transform on the data prior to fitting the model and making a prediction. Using the pipeline will ensure that the transform is always performed correctly on input data. First, we can define the model as a pipeline.

```
...
# define model to evaluate
model = SVC(gamma='scale', class_weight='balanced')
# power transform then fit model
pipeline = Pipeline(steps=[('t',PowerTransformer()), ('m',model)])
```

Listing 29.24: Example of defining the final model.

Once defined, we can fit it on the entire training dataset.

```
...
```

```
# fit the model
pipeline.fit(X, y)
```

Listing 29.25: Example of fitting the final model.

Once fit, we can use it to make predictions for new data by calling the `predict()` function. This will return the class label of 0 for *no cancer*, or 1 for *cancer*. For example:

```
...
# define a row of data
row = [...]
# make prediction
yhat = model.predict([row])
```

Listing 29.26: Example of making a prediction with the fit final model.

To demonstrate this, we can use the fit model to make some predictions of labels for a few cases where we know if the case is a no cancer or cancer. The complete example is listed below.

```
# fit a model and make predictions for the mammography dataset
from pandas import read_csv
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import PowerTransformer
from sklearn.svm import SVC
from sklearn.pipeline import Pipeline

# load the dataset
def load_dataset(full_path):
    # load the dataset as a numpy array
    data = read_csv(full_path, header=None)
    # retrieve numpy array
    data = data.values
    # split into input and output elements
    X, y = data[:, :-1], data[:, -1]
    # label encode the target variable to have the classes 0 and 1
    y = LabelEncoder().fit_transform(y)
    return X, y

# define the location of the dataset
full_path = 'mammography.csv'
# load the dataset
X, y = load_dataset(full_path)
# define model to evaluate
model = SVC(gamma='scale', class_weight='balanced')
# power transform then fit model
pipeline = Pipeline(steps=[('t',PowerTransformer()), ('m',model)])
# fit the model
pipeline.fit(X, y)
# evaluate on some no cancer cases (known class 0)
print('No Cancer:')
data = [[0.23001961, 5.0725783, -0.27606055, 0.83244412, -0.37786573, 0.4803223],
        [0.15549112, -0.16939038, 0.67065219, -0.85955255, -0.37786573, -0.94572324],
        [-0.78441482, -0.44365372, 5.6747053, -0.85955255, -0.37786573, -0.94572324]]
for row in data:
    # make prediction
    yhat = pipeline.predict([row])
    # get the label
```

```

label = yhat[0]
# summarize
print('>Predicted=%d (expected 0)' % (label))
# evaluate on some cancer (known class 1)
print('Cancer:')
data = [[2.0158239,0.15353258,-0.32114211,2.1923706,-0.37786573,0.96176503],
        [2.3191888,0.72860087,-0.50146835,-0.85955255,-0.37786573,-0.94572324],
        [0.19224721,-0.2003556,-0.230979,1.2003796,2.2620867,1.132403]]
for row in data:
    # make prediction
    yhat = pipeline.predict([row])
    # get the label
    label = yhat[0]
    # summarize
    print('>Predicted=%d (expected 1)' % (label))

```

Listing 29.27: Fit a model and use it to make predictions.

Running the example first fits the model on the entire training dataset. Then the fit model used to predict the label of no cancer cases is chosen from the dataset file. We can see that all cases are correctly predicted. Then some cases of actual cancer are used as input to the model and the label is predicted. As we might have hoped, the correct labels are predicted for all cases.

```

No Cancer:
>Predicted=0 (expected 0)
>Predicted=0 (expected 0)
>Predicted=0 (expected 0)
Cancer:
>Predicted=1 (expected 1)
>Predicted=1 (expected 1)
>Predicted=1 (expected 1)

```

Listing 29.28: Example output from fitting a model and use it to make predictions.

29.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

29.7.1 Papers

- *Comparative Evaluation Of Pattern Recognition Techniques For Detection Of Microcalcifications In Mammography*, 1993.
<https://www.worldscientific.com/doi/abs/10.1142/S0218001493000698>
- *SMOTE: Synthetic Minority Over-sampling Technique*, 2002.
<https://arxiv.org/abs/1106.1813>

29.7.2 APIs

- `sklearn.metrics.roc_auc_score` API.
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_auc_score.html

- `sklearn.dummy.DummyClassifier` API.
<https://scikit-learn.org/stable/modules/generated/sklearn.dummy.DummyClassifier.html>
- `sklearn.svm.SVC` API.
<https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html>

29.7.3 Dataset

- Mammography Dataset.
<https://raw.githubusercontent.com/jbrownlee/Datasets/master/mammography.csv>
- Mammography Dataset Description.
<https://raw.githubusercontent.com/jbrownlee/Datasets/master/mammography.names>

29.8 Summary

In this tutorial, you discovered how to develop and evaluate models for imbalanced mammography cancer classification dataset. Specifically, you learned:

- How to load and explore the dataset and generate ideas for data preparation and model selection.
- How to evaluate a suite of machine learning models and improve their performance with data cost-sensitive techniques.
- How to fit a final model and use it to predict class labels for specific cases.

29.8.1 Next

In the next tutorial, you will discover how to systematically work through the phoneme imbalanced classification dataset.

Chapter 30

Project: Phoneme Classification

Many binary classification tasks do not have an equal number of examples from each class, e.g. the class distribution is skewed or imbalanced. Nevertheless, accuracy is equally important in both classes.

An example is the classification of vowel sounds from European languages as either nasal or oral in speech recognition where there are many more examples of nasal than oral vowels. Classification accuracy is important for both classes, although accuracy as a metric cannot be used directly because of the outcome class imbalance. Additionally, data sampling techniques may be required to transform the training dataset to make it more balanced when fitting machine learning algorithms. In this tutorial, you will discover how to develop and evaluate models for imbalanced binary classification of nasal and oral phonemes. After completing this tutorial, you will know:

- How to load and explore the dataset and generate ideas for data preparation and model selection.
- How to evaluate a suite of machine learning models and improve their performance with data oversampling techniques.
- How to fit a final model and use it to predict class labels for specific cases.

Let's get started.

Note: This chapter makes use of the imbalanced-learn library. See Appendix [B](#) for installation instructions, if needed.

30.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Phoneme Dataset
2. Explore the Dataset
3. Model Test and Baseline Result
4. Evaluate Models
5. Make Predictions on New Data

30.2 Phoneme Dataset

In this project, we will use a standard imbalanced machine learning dataset referred to as the *Phoneme* dataset. This dataset is credited to the ESPRIT (European Strategic Program on Research in Information Technology) project titled *ROARS* (Robust Analytical Speech Recognition System) and described in progress reports and technical reports from that project.

The goal of the ROARS project is to increase the robustness of an existing analytical speech recognition system (i.e., one using knowledge about syllables, phonemes and phonetic features), and to use it as part of a speech understanding system with connected words and dialogue capability. This system will be evaluated for a specific application in two European languages

— *ESPRIT: The European Strategic Programme for Research and development in Information Technology.*

The goal of the dataset was to distinguish between nasal and oral vowels. Vowel sounds were spoken and recorded to digital files. Then audio features were automatically extracted from each sound.

Five different attributes were chosen to characterize each vowel: they are the amplitudes of the five first harmonics AHi, normalised by the total energy Ene (integrated on all the frequencies): AHi/Ene. Each harmonic is signed: positive when it corresponds to a local maximum of the spectrum and negative otherwise.

— *Phoneme Dataset Description.*

There are two classes for the two types of sounds; they are:

- **Class 0:** Nasal Vowels (majority class).
- **Class 1:** Oral Vowels (minority class).

Next, let's take a closer look at the data.

30.3 Explore the Dataset

The Phoneme dataset is a widely used standard machine learning dataset, used to explore and demonstrate many techniques designed specifically for imbalanced classification. One example is the popular SMOTE data oversampling technique. First, download the dataset and save it in your current working directory with the name `phoneme.csv`.

- Download Phoneme Dataset (`phoneme.csv`). ¹

Review the contents of the file. The first few lines of the file should look as follows:

¹<https://raw.githubusercontent.com/jbrownlee/Datasets/master/phoneme.csv>

```
1.24,0.875,-0.205,-0.078,0.067,0
0.268,1.352,1.035,-0.332,0.217,0
1.567,0.867,1.3,1.041,0.559,0
0.279,0.99,2.555,-0.738,0.0,0
0.307,1.272,2.656,-0.946,-0.467,0
...
```

Listing 30.1: Example of rows of data from the phoneme dataset.

We can see that the given input variables are numeric and class labels are 0 and 1 for nasal and oral respectively. The dataset can be loaded as a `DataFrame` using the `read_csv()` Pandas function, specifying the location and the fact that there is no header line.

```
...
# define the dataset location
filename = 'phoneme.csv'
# load the csv file as a data frame
dataframe = read_csv(filename, header=None)
```

Listing 30.2: Example of loading the phoneme dataset.

Once loaded, we can summarize the number of rows and columns by printing the shape of the `DataFrame`.

```
...
# summarize the shape of the dataset
print(dataframe.shape)
```

Listing 30.3: Example of summarizing the shape of the loaded dataset.

We can also summarize the number of examples in each class using the `Counter` object.

```
...
# summarize the class distribution
target = dataframe.values[:, -1]
counter = Counter(target)
for k, v in counter.items():
    per = v / len(target) * 100
    print('Class=%s, Count=%d, Percentage=%.3f%%' % (k, v, per))
```

Listing 30.4: Example of summarizing the class distribution for the loaded dataset.

Tying this together, the complete example of loading and summarizing the dataset is listed below.

```
# load and summarize the dataset
from pandas import read_csv
from collections import Counter
# define the dataset location
filename = 'phoneme.csv'
# load the csv file as a data frame
dataframe = read_csv(filename, header=None)
# summarize the shape of the dataset
print(dataframe.shape)
# summarize the class distribution
target = dataframe.values[:, -1]
counter = Counter(target)
for k, v in counter.items():
```

```
per = v / len(target) * 100
print('Class=%s, Count=%d, Percentage=%.3f%%' % (k, v, per))
```

Listing 30.5: Load and summarize the phoneme dataset.

Running the example first loads the dataset and confirms the number of rows and columns, that is 5,404 rows and five input variables and one target variable. The class distribution is then summarized, confirming a modest class imbalance with approximately 70 percent for the majority class (*nasal*) and approximately 30 percent for the minority class (*oral*).

```
(5404, 6)
Class=0.0, Count=3818, Percentage=70.651%
Class=1.0, Count=1586, Percentage=29.349%
```

Listing 30.6: Example output from loading and summarizing the phoneme dataset.

We can also take a look at the distribution of the five numerical input variables by creating a histogram for each.

The complete example is listed below.

```
# create histograms of numeric input variables
from pandas import read_csv
from matplotlib import pyplot
# define the dataset location
filename = 'phoneme.csv'
# load the csv file as a data frame
df = read_csv(filename, header=None)
# histograms of all variables
df.hist()
pyplot.show()
```

Listing 30.7: Plot histograms of the phoneme dataset.

Running the example creates the figure with one histogram subplot for each of the five numerical input variables in the dataset, as well as the numerical class label. We can see that the variables have differing scales, although most appear to have a Gaussian or Gaussian-like distribution. Depending on the choice of modeling algorithms, we would expect scaling the distributions to the same range to be useful, and perhaps the use of some power transforms.

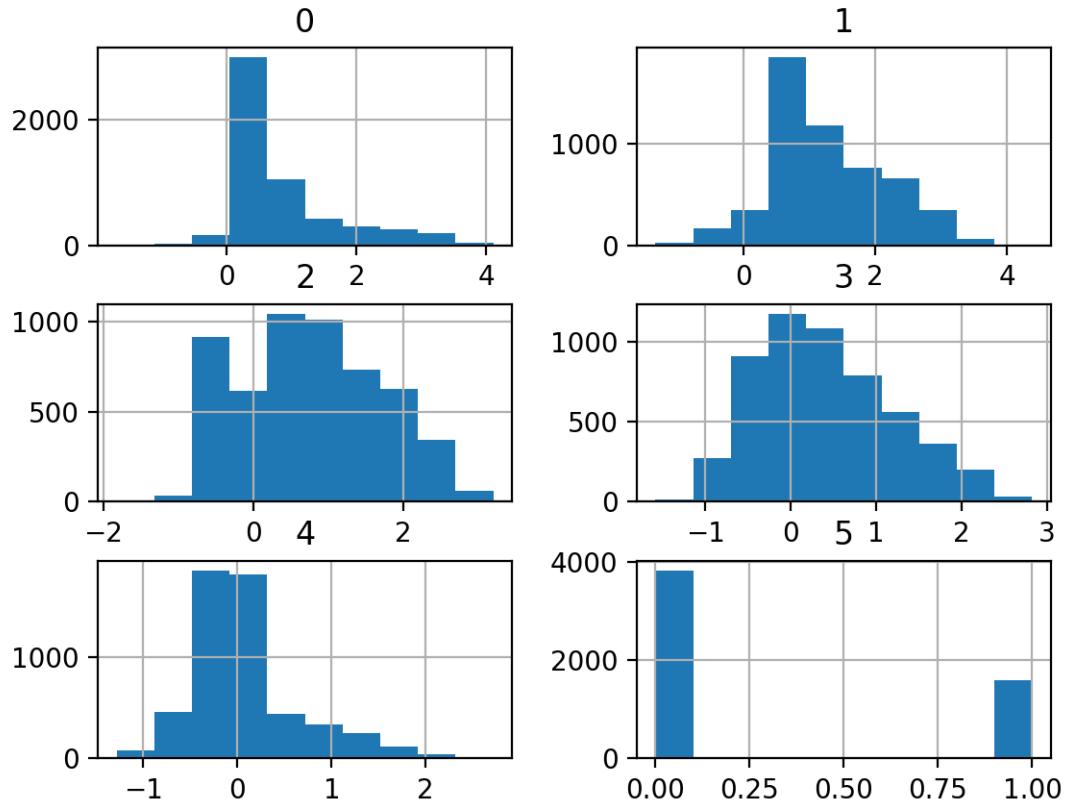


Figure 30.1: Histogram Plots of the Variables for the Phoneme Dataset.

We can also create a scatter plot for each pair of input variables, called a scatter plot matrix. This can be helpful to see if any variables relate to each other or change in the same direction, i.e. are correlated. We can also color the dots of each scatter plot according to the class label. In this case, the majority class (nasal) will be mapped to blue dots and the minority class (oral) will be mapped to red dots. The complete example is listed below.

```
# create pairwise scatter plots of numeric input variables
from pandas import read_csv
from pandas import DataFrame
from pandas.plotting import scatter_matrix
from matplotlib import pyplot
# define the dataset location
filename = 'phoneme.csv'
# load the csv file as a data frame
df = read_csv(filename, header=None)
# define a mapping of class values to colors
color_dict = {0:'blue', 1:'red'}
# map each row to a color based on the class value
colors = [color_dict[x] for x in df.values[:, -1]]
# drop the target variable
inputs = DataFrame(df.values[:, :-1])
# pairwise scatter plots of all numerical variables
scatter_matrix(inputs, diagonal='kde', color=colors)
```

```
pyplot.show()
```

Listing 30.8: Scatter plot matrix of the phoneme dataset.

Running the example creates a figure showing the scatter plot matrix, with five plots by five plots, comparing each of the five numerical input variables with each other. The diagonal of the matrix shows the density distribution of each variable. Each pairing appears twice, both above and below the top-left to bottom-right diagonal, providing two ways to review the same variable interactions. We can see that the distributions for many variables do differ for the two class labels, suggesting that some reasonable discrimination between the classes will be feasible.

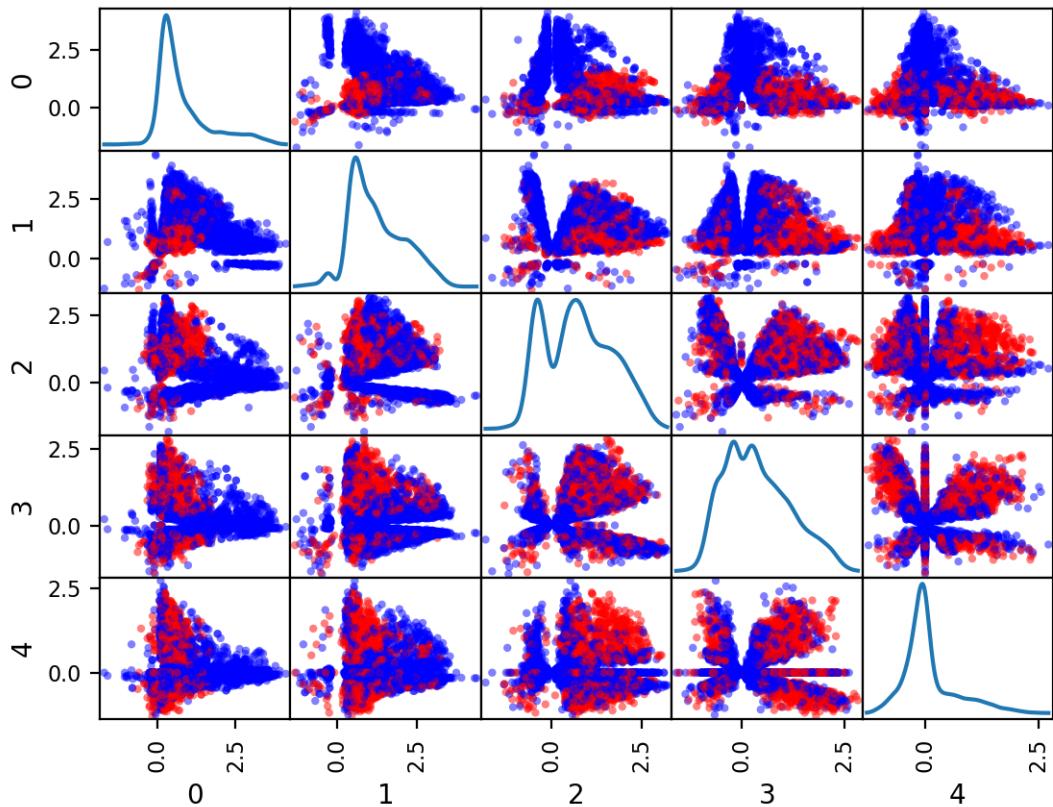


Figure 30.2: Scatter Plot Matrix by Class for the Numerical Input Variables in the Phoneme Dataset.

Now that we have reviewed the dataset, let's look at developing a test harness for evaluating candidate models.

30.4 Model Test and Baseline Result

We will evaluate candidate models using repeated stratified k -fold cross-validation. The k -fold cross-validation procedure provides a good general estimate of model performance that is not too optimistically biased, at least compared to a single train-test split. We will use $k=10$, meaning

each fold will contain about $\frac{5404}{10}$ or about 540 examples. Stratified means that each fold will contain the same mixture of examples by class, that is about 70 percent to 30 percent nasal to oral vowels. Repetition indicates that the evaluation process will be performed multiple times to help avoid fluke results and better capture the variance of the chosen model. We will use three repeats.

This means a single model will be fit and evaluated 10×3 (30) times and the mean and standard deviation of these runs will be reported. This can be achieved using the `RepeatedStratifiedKFold` scikit-learn class. Class labels will be predicted and both class labels are equally important. Therefore, we will select a metric that quantifies the performance of a model on both classes separately. You may remember that the sensitivity is a measure of the accuracy for the positive class and specificity is a measure of the accuracy of the negative class.

The G-mean seeks a balance of these scores, the geometric mean, where poor performance for one or the other results in a low G-mean score (for more on the G-mean, see Chapter 4). We can calculate the G-mean for a set of predictions made by a model using the `geometric_mean_score()` function provided by the imbalanced-learn library. We can define a function to load the dataset and split the columns into input and output variables. The `load_dataset()` function below implements this.

```
# load the dataset
def load_dataset(full_path):
    # load the dataset as a numpy array
    data = read_csv(full_path, header=None)
    # retrieve numpy array
    data = data.values
    # split into input and output elements
    X, y = data[:, :-1], data[:, -1]
    return X, y
```

Listing 30.9: Example of a function for loading and preparing the dataset for modeling.

We can then define a function that will evaluate a given model on the dataset and return a list of G-mean scores for each fold and repeat. The `evaluate_model()` function below implements this, taking the dataset and model as arguments and returning the list of scores.

```
# evaluate a model
def evaluate_model(X, y, model):
    # define evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # define the model evaluation the metric
    metric = make_scorer(geometric_mean_score)
    # evaluate model
    scores = cross_val_score(model, X, y, scoring=metric, cv=cv, n_jobs=-1)
    return scores
```

Listing 30.10: Example of a function for evaluating a defined model.

Finally, we can evaluate a baseline model on the dataset using this test harness. A model that predicts the majority class label (0) or the minority class label (1) for all cases will result in a G-mean of zero. As such, a good default strategy would be to randomly predict one class label or another with a 50 percent probability and aim for a G-mean of about 0.5. This can be achieved using the `DummyClassifier` class from the scikit-learn library and setting the `strategy` argument to ‘uniform’.

```
...
# define the reference model
model = DummyClassifier(strategy='uniform')
```

Listing 30.11: Example of defining the baseline model.

Once the model is evaluated, we can report the mean and standard deviation of the G-mean scores directly.

```
...
# evaluate the model
scores = evaluate_model(X, y, model)
# summarize performance
print('Mean G-mean: %.3f (%.3f)' % (mean(scores), std(scores)))
```

Listing 30.12: Example of evaluating the model and summarizing the performance.

Tying this together, the complete example of loading the dataset, evaluating a baseline model, and reporting the performance is listed below.

```
# test harness and baseline model evaluation
from collections import Counter
from numpy import mean
from numpy import std
from pandas import read_csv
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from imblearn.metrics import geometric_mean_score
from sklearn.metrics import make_scorer
from sklearn.dummy import DummyClassifier

# load the dataset
def load_dataset(full_path):
    # load the dataset as a numpy array
    data = read_csv(full_path, header=None)
    # retrieve numpy array
    data = data.values
    # split into input and output elements
    X, y = data[:, :-1], data[:, -1]
    return X, y

# evaluate a model
def evaluate_model(X, y, model):
    # define evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # define the model evaluation metric
    metric = make_scorer(geometric_mean_score)
    # evaluate model
    scores = cross_val_score(model, X, y, scoring=metric, cv=cv, n_jobs=-1)
    return scores

# define the location of the dataset
full_path = 'phoneme.csv'
# load the dataset
X, y = load_dataset(full_path)
# summarize the loaded dataset
```

```

print(X.shape, y.shape, Counter(y))
# define the reference model
model = DummyClassifier(strategy='uniform')
# evaluate the model
scores = evaluate_model(X, y, model)
# summarize performance
print('Mean G-Mean: %.3f (%.3f)' % (mean(scores), std(scores)))

```

Listing 30.13: Calculate the baseline performance for the phoneme dataset.

Running the example first loads and summarizes the dataset. We can see that we have the correct number of rows loaded and that we have five audio-derived input variables. Next, the average of the G-mean scores is reported.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the baseline algorithm achieves a G-mean of about 0.509, close to the theoretical maximum of 0.5. This score provides a lower limit on model skill; any model that achieves an average G-mean above about 0.509 (or really above 0.5) has skill, whereas models that achieve a score below this value do not have skill on this dataset.

```
(5404, 5) (5404,) Counter({0.0: 3818, 1.0: 1586})
Mean G-mean: 0.509 (0.020)
```

Listing 30.14: Example output from calculating the baseline performance for the phoneme dataset.

Now that we have a test harness and a baseline in performance, we can begin to evaluate some models on this dataset.

30.5 Evaluate Models

In this section, we will evaluate a suite of different techniques on the dataset using the test harness developed in the previous section. The goal is to both demonstrate how to work through the problem systematically and to demonstrate the capability of some techniques designed for imbalanced classification problems. The reported performance is good, but not highly optimized (e.g. hyperparameters are not tuned).

30.5.1 Evaluate Machine Learning Algorithms

Let's start by evaluating a mixture of machine learning models on the dataset. It can be a good idea to spot-check a suite of different linear and nonlinear algorithms on a dataset to quickly flush out what works well and deserves further attention, and what doesn't. We will evaluate the following machine learning models on the phoneme dataset:

- Logistic Regression (LR)
- Support Vector Machine (SVM)
- Bagged Decision Trees (BAG)

- Random Forest (RF)
- Extra Trees (ET)

We will use mostly default model hyperparameters, with the exception of the number of trees in the ensemble algorithms, which we will set to a reasonable default of 1,000. We will define each model in turn and add them to a list so that we can evaluate them sequentially. The `get_models()` function below defines the list of models for evaluation, as well as a list of model short names for plotting the results later.

```
# define models to test
def get_models():
    models, names = list(), list()
    # LR
    models.append(LogisticRegression(solver='lbfgs'))
    names.append('LR')
    # SVM
    models.append(SVC(gamma='scale'))
    names.append('SVM')
    # Bagging
    models.append(BaggingClassifier(n_estimators=1000))
    names.append('BAG')
    # RF
    models.append(RandomForestClassifier(n_estimators=1000))
    names.append('RF')
    # ET
    models.append(ExtraTreesClassifier(n_estimators=1000))
    names.append('ET')
    return models, names
```

Listing 30.15: Example of a function for defining the models to evaluate.

We can then enumerate the list of models in turn and evaluate each, reporting the mean G-mean and storing the scores for later plotting.

```
...
# define models
models, names = get_models()
results = list()
# evaluate each model
for i in range(len(models)):
    # evaluate the model and store results
    scores = evaluate_model(X, y, models[i])
    results.append(scores)
    # summarize and store
    print('>%s %.3f (%.3f)' % (names[i], mean(scores), std(scores)))
```

Listing 30.16: Example of evaluating the defined models.

At the end of the run, we can plot each sample of scores as a box and whisker plot with the same scale so that we can directly compare the distributions.

```
...
# plot the results
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

Listing 30.17: Example of summarizing performance using box and whisker plots.

Tying this all together, the complete example of evaluating a suite of machine learning algorithms on the phoneme dataset is listed below.

```
# spot check machine learning algorithms on the phoneme dataset
from numpy import mean
from numpy import std
from pandas import read_csv
from matplotlib import pyplot
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from imblearn.metrics import geometric_mean_score
from sklearn.metrics import make_scorer
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.ensemble import BaggingClassifier

# load the dataset
def load_dataset(full_path):
    # load the dataset as a numpy array
    data = read_csv(full_path, header=None)
    # retrieve numpy array
    data = data.values
    # split into input and output elements
    X, y = data[:, :-1], data[:, -1]
    return X, y

# evaluate a model
def evaluate_model(X, y, model):
    # define evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # define the model evaluation metric
    metric = make_scorer(geometric_mean_score)
    # evaluate model
    scores = cross_val_score(model, X, y, scoring=metric, cv=cv, n_jobs=-1)
    return scores

# define models to test
def get_models():
    models, names = list(), list()
    # LR
    models.append(LogisticRegression(solver='lbfgs'))
    names.append('LR')
    # SVM
    models.append(SVC(gamma='scale'))
    names.append('SVM')
    # Bagging
    models.append(BaggingClassifier(n_estimators=1000))
    names.append('BAG')
    # RF
    models.append(RandomForestClassifier(n_estimators=1000))
    names.append('RF')
    # ET
    models.append(ExtraTreesClassifier(n_estimators=1000))
    names.append('ET')
```

```

    return models, names

# define the location of the dataset
full_path = 'phoneme.csv'
# load the dataset
X, y = load_dataset(full_path)
# define models
models, names = get_models()
results = list()
# evaluate each model
for i in range(len(models)):
    # evaluate the model and store results
    scores = evaluate_model(X, y, models[i])
    results.append(scores)
    # summarize and store
    print('>%s %.3f (%.3f)' % (names[i], mean(scores), std(scores)))
# plot the results
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()

```

Listing 30.18: Calculate the performance of machine learning models on the phoneme dataset.

Running the example evaluates each algorithm in turn and reports the mean and standard deviation G-mean.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that all of the tested algorithms have skill, achieving a G-mean above the default of 0.5. The results suggest that the ensemble of decision tree algorithms perform better on this dataset with perhaps Extra Trees (ET) performing the best with a G-mean of about 0.896.

```

>LR 0.637 (0.023)
>SVM 0.801 (0.022)
>BAG 0.888 (0.017)
>RF 0.892 (0.018)
>ET 0.896 (0.017)

```

Listing 30.19: Example output from calculating the performance of machine learning models on the phoneme dataset.

A figure is created showing one box and whisker plot for each algorithm's sample of results. The box shows the middle 50 percent of the data, the orange line in the middle of each box shows the median of the sample, and the green triangle in each box shows the mean of the sample. We can see that all three ensembles of trees algorithms (BAG, RF, and ET) have a tight distribution and a mean and median that closely align, perhaps suggesting a non-skewed and Gaussian distribution of scores, e.g. stable.

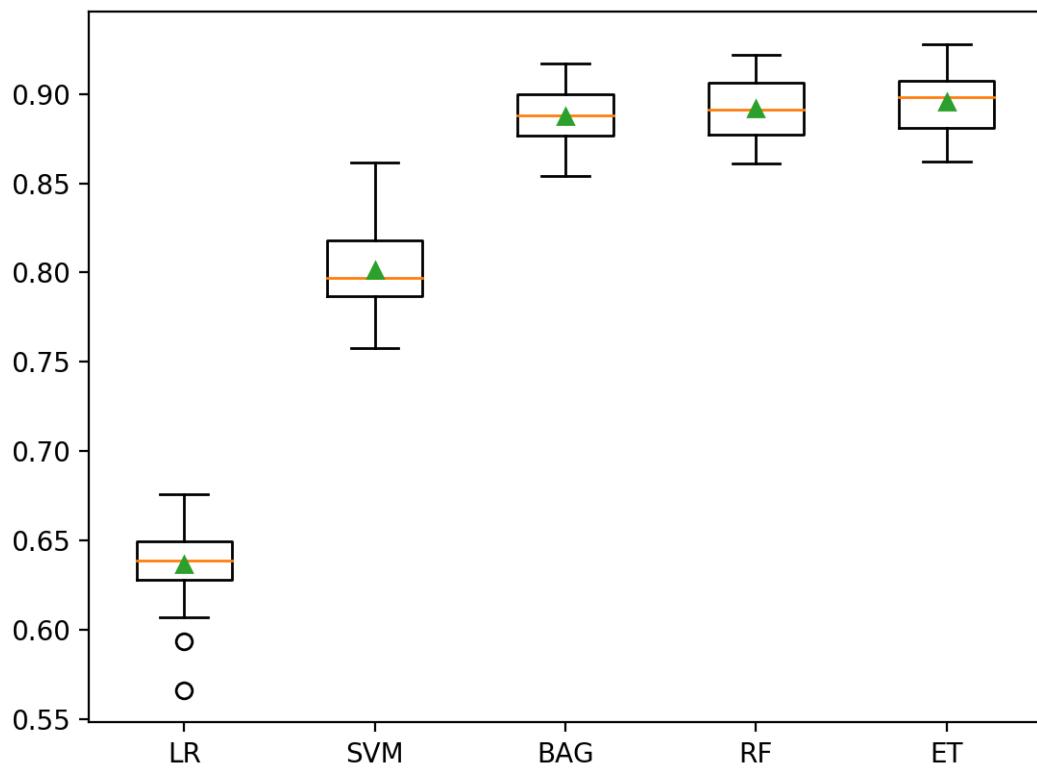


Figure 30.3: Box and Whisker Plot of Machine Learning Models on the Imbalanced Phoneme Dataset.

Now that we have a good first set of results, let's see if we can improve them with data oversampling methods.

30.5.2 Evaluate Data Oversampling Algorithms

Data sampling provides a way to better prepare the imbalanced training dataset prior to fitting a model. The simplest oversampling technique is to duplicate examples in the minority class, called random oversampling. Perhaps the most popular oversampling method is the SMOTE oversampling technique for creating new synthetic examples for the minority class. We will test five different oversampling methods; specifically:

- Random Oversampling (ROS)
- SMOTE
- BorderLine SMOTE (BLSMOTE)
- SVM SMOTE
- ADASYN

Each technique will be tested with the best performing algorithm from the previous section, specifically Extra Trees. We will use the default hyperparameters for each oversampling algorithm, which will oversample the minority class to have the same number of examples as the majority class in the training dataset. The expectation is that each oversampling technique will result in a lift in performance compared to the algorithm without oversampling with the smallest lift provided by Random Oversampling and perhaps the best lift provided by SMOTE or one of its variations. We can update the `get_models()` function to return lists of oversampling algorithms to evaluate; for example:

```
# define oversampling models to test
def get_models():
    models, names = list(), list()
    # RandomOverSampler
    models.append(RandomOverSampler())
    names.append('ROS')
    # SMOTE
    models.append(SMOTE())
    names.append('SMOTE')
    # BorderlineSMOTE
    models.append(BorderlineSMOTE())
    names.append('BLSMOTE')
    # SVMSMOTE
    models.append(SVMSMOTE())
    names.append('SVMSMOTE')
    # ADASYN
    models.append(ADASYN())
    names.append('ADASYN')
    return models, names
```

Listing 30.20: Example of a function for defining the models to evaluate.

We can then enumerate each and create a `Pipeline` from the imbalanced-learn library that is aware of how to oversample a training dataset. This will ensure that the training dataset within the cross-validation model evaluation is sampled correctly, without data leakage that could result in an optimistic evaluation of model performance.

First, we will normalize the input variables because most oversampling techniques will make use of a nearest neighbor algorithm and it is important that all variables have the same scale when using this technique. This will be followed by a given oversampling algorithm, then ending with the Extra Trees algorithm that will be fit on the oversampled training dataset.

```
...
# define the model
model = ExtraTreesClassifier(n_estimators=1000)
# define the pipeline steps
steps = [('s', MinMaxScaler()), ('o', models[i]), ('m', model)]
# define the pipeline
pipeline = Pipeline(steps=steps)
# evaluate the model and store results
scores = evaluate_model(X, y, pipeline)
```

Listing 30.21: Example of defining a pipeline for a data sampling model.

Tying this together, the complete example of evaluating oversampling algorithms with Extra Trees on the phoneme dataset is listed below.

```
# data oversampling algorithms on the phoneme imbalanced dataset
from numpy import mean
from numpy import std
from pandas import read_csv
from matplotlib import pyplot
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from imblearn.metrics import geometric_mean_score
from sklearn.metrics import make_scorer
from sklearn.ensemble import ExtraTreesClassifier
from imblearn.over_sampling import RandomOverSampler
from imblearn.over_sampling import SMOTE
from imblearn.over_sampling import BorderlineSMOTE
from imblearn.over_sampling import SVMSMOTE
from imblearn.over_sampling import ADASYN
from imblearn.pipeline import Pipeline

# load the dataset
def load_dataset(full_path):
    # load the dataset as a numpy array
    data = read_csv(full_path, header=None)
    # retrieve numpy array
    data = data.values
    # split into input and output elements
    X, y = data[:, :-1], data[:, -1]
    return X, y

# evaluate a model
def evaluate_model(X, y, model):
    # define evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    # define the model evaluation metric
    metric = make_scorer(geometric_mean_score)
    # evaluate model
    scores = cross_val_score(model, X, y, scoring=metric, cv=cv, n_jobs=-1)
    return scores

# define oversampling models to test
def get_models():
    models, names = list(), list()
    # RandomOverSampler
    models.append(RandomOverSampler())
    names.append('ROS')
    # SMOTE
    models.append(SMOTE())
    names.append('SMOTE')
    # BorderlineSMOTE
    models.append(BorderlineSMOTE())
    names.append('BLSMOTE')
    # SVMSMOTE
    models.append(SVMSMOTE())
    names.append('SVMSMOTE')
    # ADASYN
    models.append(ADASYN())
    names.append('ADASYN')
```

```

names.append('ADASYN')
return models, names

# define the location of the dataset
full_path = 'phoneme.csv'
# load the dataset
X, y = load_dataset(full_path)
# define models
models, names = get_models()
results = list()
# evaluate each model
for i in range(len(models)):
    # define the model
    model = ExtraTreesClassifier(n_estimators=1000)
    # define the pipeline steps
    steps = [('s', MinMaxScaler()), ('o', models[i]), ('m', model)]
    # define the pipeline
    pipeline = Pipeline(steps=steps)
    # evaluate the model and store results
    scores = evaluate_model(X, y, pipeline)
    results.append(scores)
    # summarize and store
    print('>%s %.3f (%.3f)' % (names[i], mean(scores), std(scores)))
# plot the results
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()

```

Listing 30.22: Calculate the performance of data sampling models on the phoneme dataset.

Running the example evaluates each oversampling method with the Extra Trees model on the dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, as we expected, each oversampling technique resulted in a lift in performance for the ET algorithm without any oversampling (0.896), except the random oversampling technique. The results suggest that the modified versions of SMOTE and ADASYN performed better than default SMOTE, and in this case, ADASYN achieved the best G-mean score of 0.910.

```

>ROS 0.894 (0.018)
>SMOTE 0.906 (0.015)
>BLSMOTE 0.909 (0.013)
>SVMSMOTE 0.909 (0.014)
>ADASYN 0.910 (0.013)

```

Listing 30.23: Example output from calculating the performance of data sampling models on the phoneme dataset.

The distribution of results can be compared with box and whisker plots. We can see the distributions all roughly have the same tight distribution and that the difference in means of the results can be used to select a model.

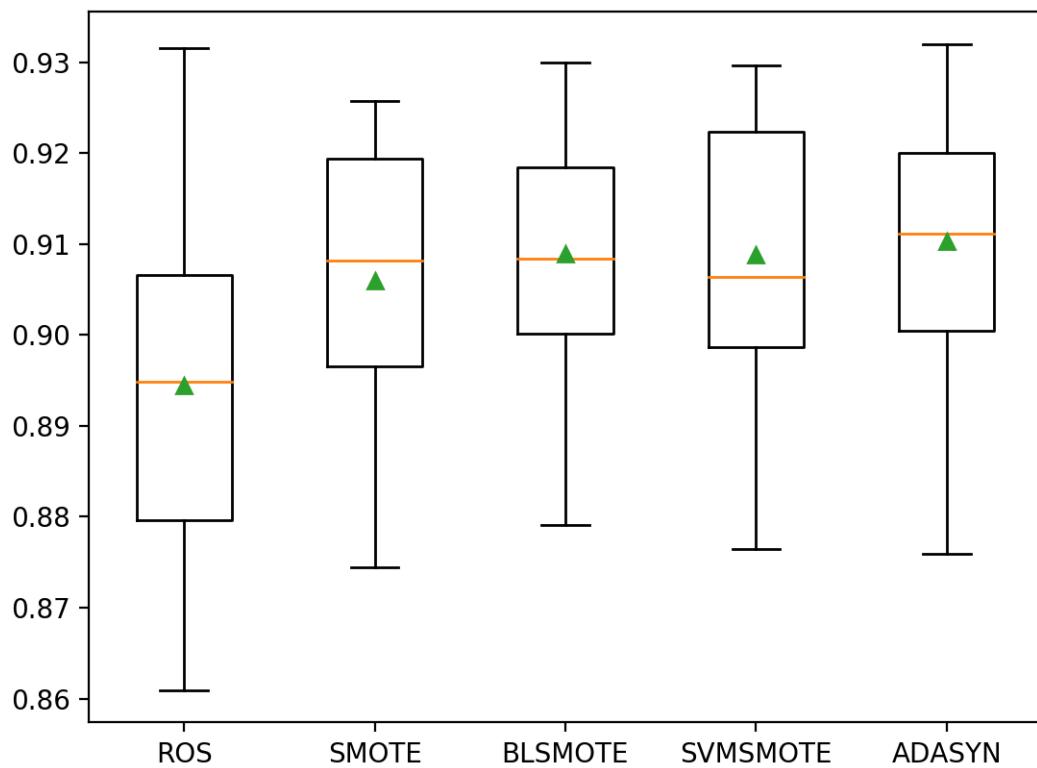


Figure 30.4: Box and Whisker Plot of Extra Trees Models With Data Oversampling on the Imbalanced Phoneme Dataset.

Next, let's see how we might use a final model to make predictions on new data.

30.6 Make Prediction on New Data

In this section, we will fit a final model and use it to make predictions on single rows of data. We will use the ADASYN oversampled version of the Extra Trees model as the final model and a normalization scaling on the data prior to fitting the model and making a prediction. Using the pipeline will ensure that the transform is always performed correctly. First, we can define the model as a pipeline.

```
...
# define the model
model = ExtraTreesClassifier(n_estimators=1000)
# define the pipeline steps
steps = [('s', MinMaxScaler()), ('o', ADASYN()), ('m', model)]
# define the pipeline
pipeline = Pipeline(steps=steps)
```

Listing 30.24: Example of defining the final model.

Once defined, we can fit it on the entire training dataset.

```
...
# fit the model
pipeline.fit(X, y)
```

Listing 30.25: Example of fitting the final model.

Once fit, we can use it to make predictions for new data by calling the `predict()` function. This will return the class label of 0 for *nasal*, or 1 for *oral*. For example:

```
...
# define a row of data
row = [...]
# make prediction
yhat = pipeline.predict([row])
```

Listing 30.26: Example of making a prediction with the fit final model.

To demonstrate this, we can use the fit model to make some predictions of labels for a few cases where we know if the case is nasal or oral. The complete example is listed below.

```
# fit a model and make predictions for the phoneme dataset
from pandas import read_csv
from sklearn.preprocessing import MinMaxScaler
from imblearn.over_sampling import ADASYN
from sklearn.ensemble import ExtraTreesClassifier
from imblearn.pipeline import Pipeline

# load the dataset
def load_dataset(full_path):
    # load the dataset as a numpy array
    data = read_csv(full_path, header=None)
    # retrieve numpy array
    data = data.values
    # split into input and output elements
    X, y = data[:, :-1], data[:, -1]
    return X, y

# define the location of the dataset
full_path = 'phoneme.csv'
# load the dataset
X, y = load_dataset(full_path)
# define the model
model = ExtraTreesClassifier(n_estimators=1000)
# define the pipeline steps
steps = [('s', MinMaxScaler()), ('o', ADASYN()), ('m', model)]
# define the pipeline
pipeline = Pipeline(steps=steps)
# fit the model
pipeline.fit(X, y)
# evaluate on some nasal cases (known class 0)
print('Nasal:')
data = [[1.24, 0.875, -0.205, -0.078, 0.067],
        [0.268, 1.352, 1.035, -0.332, 0.217],
        [1.567, 0.867, 1.3, 1.041, 0.559]]
for row in data:
    # make prediction
```

```

yhat = pipeline.predict([row])
# get the label
label = yhat[0]
# summarize
print('>Predicted=%d (expected 0)' % (label))
# evaluate on some oral cases (known class 1)
print('Oral:')
data = [[0.125,0.548,0.795,0.836,0.0],
        [0.318,0.811,0.818,0.821,0.86],
        [0.151,0.642,1.454,1.281,-0.716]]
for row in data:
    # make prediction
    yhat = pipeline.predict([row])
    # get the label
    label = yhat[0]
    # summarize
    print('>Predicted=%d (expected 1)' % (label))

```

Listing 30.27: Fit a model and use it to make predictions.

Running the example first fits the model on the entire training dataset. Then the fit model is used to predict the label of nasal cases chosen from the dataset file. We can see that all cases are correctly predicted. Then some oral cases are used as input to the model and the label is predicted. As we might have hoped, the correct labels are predicted for all cases.

```

Nasal:
>Predicted=0 (expected 0)
>Predicted=0 (expected 0)
>Predicted=0 (expected 0)
Oral:
>Predicted=1 (expected 1)
>Predicted=1 (expected 1)
>Predicted=1 (expected 1)

```

Listing 30.28: Example output from fitting a model and use it to make predictions.

30.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

30.7.1 Papers

- *ESPRIT: The European Strategic Programme for Research and development in Information Technology.*
<https://www.aclweb.org/anthology/H91-1007.pdf>

30.7.2 APIs

- *sklearn.dummy.DummyClassifier API.*
<https://scikit-learn.org/stable/modules/generated/sklearn.dummy.DummyClassifier.html>

- *imblearn.metrics.geometric_mean_score* API.

https://imbalanced-learn.org/stable/generated/imblearn.metrics.geometric_mean_score.html

30.7.3 Dataset

- Phoneme Dataset.

<https://raw.githubusercontent.com/jbrownlee/Datasets/master/phoneme.csv>

- Phoneme Dataset Description.

<https://raw.githubusercontent.com/jbrownlee/Datasets/master/phoneme.names>

30.8 Summary

In this tutorial, you discovered how to develop and evaluate models for imbalanced binary classification of nasal and oral phonemes. Specifically, you learned:

- How to load and explore the dataset and generate ideas for data preparation and model selection.
- How to evaluate a suite of machine learning models and improve their performance with data oversampling techniques.
- How to fit a final model and use it to predict class labels for specific cases.

30.8.1 Next

This was the final tutorial in this Part. In the next Part, you will discover additional helpful resources.

Part VIII

Appendix

Appendix A

Getting Help

This is just the beginning of your journey with imbalanced classification. As you start to work on projects and expand your existing knowledge of the techniques, you may need help. This appendix points out some of the best sources to turn to.

A.1 Imbalanced Classification Books

There are a number of books dedicated to the topic of imbalanced classification. A few books that I would recommend include:

- Learning from Imbalanced Data Sets, 2018.
<https://amzn.to/307Xlva>
- Imbalanced Learning: Foundations, Algorithms, and Applications, 2013.
<https://amzn.to/32K9K6d>

A.2 Machine Learning Books

There are a number of books that introduce machine learning and classification. These can be helpful but do assume you have some background in probability and linear algebra. A few books on machine learning I would recommend include:

- Applied Predictive Modeling, 2013.
<https://amzn.to/2kXE35G>
- Machine Learning: A Probabilistic Perspective, 2012.
<https://amzn.to/2xKSTCP>
- Pattern Recognition and Machine Learning, 2006.
<https://amzn.to/2JwHE7I>

A.3 Python APIs

It is a great idea to get familiar with the Python APIs that you may use on your imbalanced classification projects. Some of the APIs I recommend studying include:

- Scikit-Learn API Reference.
<https://scikit-learn.org/stable/modules/classes.html>
- Imbalanced-Learn API Reference.
<https://imbalanced-learn.org/stable/api.html>
- Matplotlib API Reference.
<http://matplotlib.org/api/index.html>

A.4 Ask Questions About Imbalanced Classification

What if you need even more help? Below are some resources where you can ask more detailed questions and get helpful technical answers.

- Cross Validated, Machine Learning Questions and Answers.
<https://stats.stackexchange.com/>
- Stack Overflow, Programming Questions and Answers.
<https://stackoverflow.com/>

A.5 How to Ask Questions

Knowing where to get help is the first step, but you need to know how to get the most out of these resources. Below are some tips that you can use:

- Boil your question down to the simplest form. E.g. not something broad like “*my model does not work*” or “*how does x work*”.
- Search for answers before asking questions.
- Provide complete code and error messages.
- Boil your code down to the smallest possible working example that demonstrates the issue.

A.6 Contact the Author

You are not alone. If you ever have any questions about imbalanced classification or the tutorials in this book, please contact me directly. I will do my best to help.

Jason Brownlee

Jason@MachineLearningMastery.com

Appendix B

How to Setup Python on Your Workstation

It can be difficult to install a Python machine learning environment on some platforms. Python itself must be installed first and then there are many packages to install, and it can be confusing for beginners. In this tutorial, you will discover how to setup a Python machine learning development environment using Anaconda.

After completing this tutorial, you will have a working Python environment to begin learning, practicing, and developing machine learning software. These instructions are suitable for Windows, macOS, and Linux platforms. I will demonstrate them on macOS, so you may see some mac dialogs and file extensions.

B.1 Tutorial Overview

In this tutorial, we will cover the following steps:

1. Download Anaconda
2. Install Anaconda
3. Start and Update Anaconda
4. Install the Imbalanced-Learn Library
5. Install the Deep Learning Libraries
6. Install the XGBoost Library

Note: The specific versions may differ as the software and libraries are updated frequently.

B.2 Download Anaconda

In this step, we will download the Anaconda Python package for your platform. Anaconda is a free and easy-to-use environment for scientific Python.

- 1. Visit the Anaconda homepage.
<https://www.continuum.io/>
- 2. Click Anaconda from the menu and click Download to go to the download page.
<https://www.continuum.io/downloads>

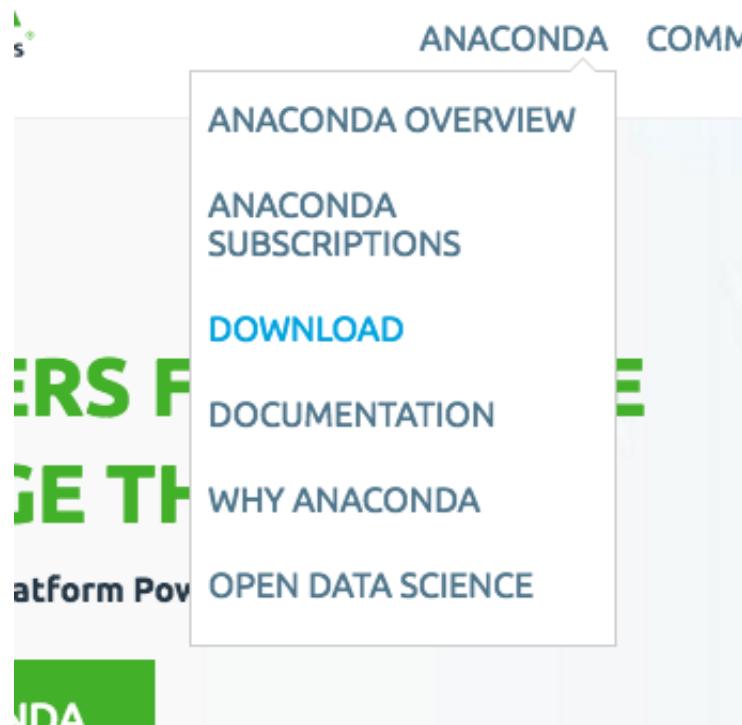


Figure B.1: Click Anaconda and Download.

- 3. Choose the download suitable for your platform (Windows, OSX, or Linux):
 - Choose Python 3.6
 - Choose the Graphical Installer

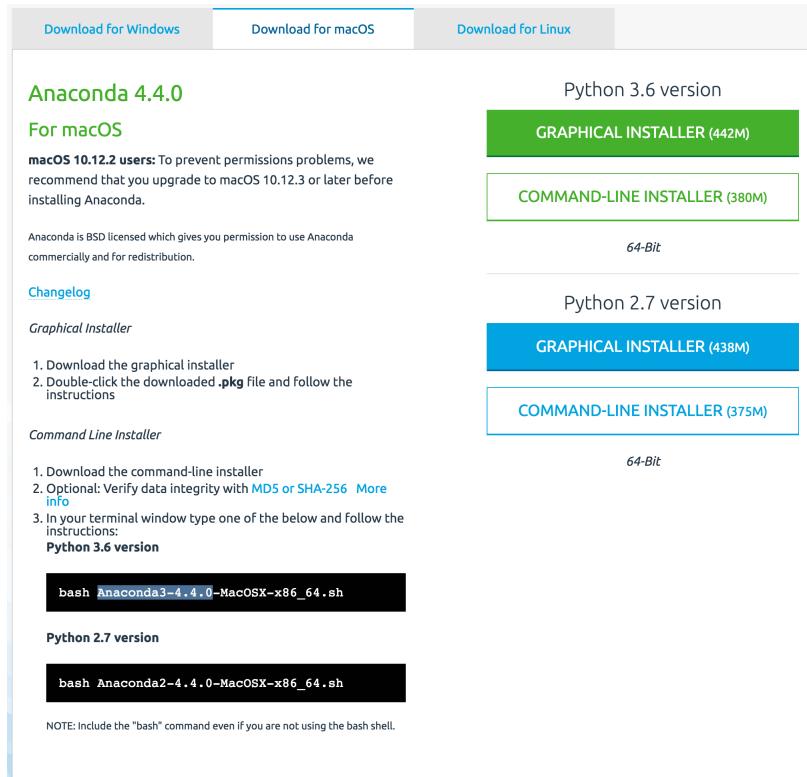


Figure B.2: Choose Anaconda Download for Your Platform.

This will download the Anaconda Python package to your workstation. I'm on macOS, so I chose the macOS version. The file is about 426 MB. You should have a file with a name like:

```
Anaconda3-4.4.0-MacOSX-x86_64.pkg
```

Listing B.1: Example filename on macOS.

B.3 Install Anaconda

In this step, we will install the Anaconda Python software on your system. This step assumes you have sufficient administrative privileges to install software on your system.

- 1. Double click the downloaded file.
- 2. Follow the installation wizard.

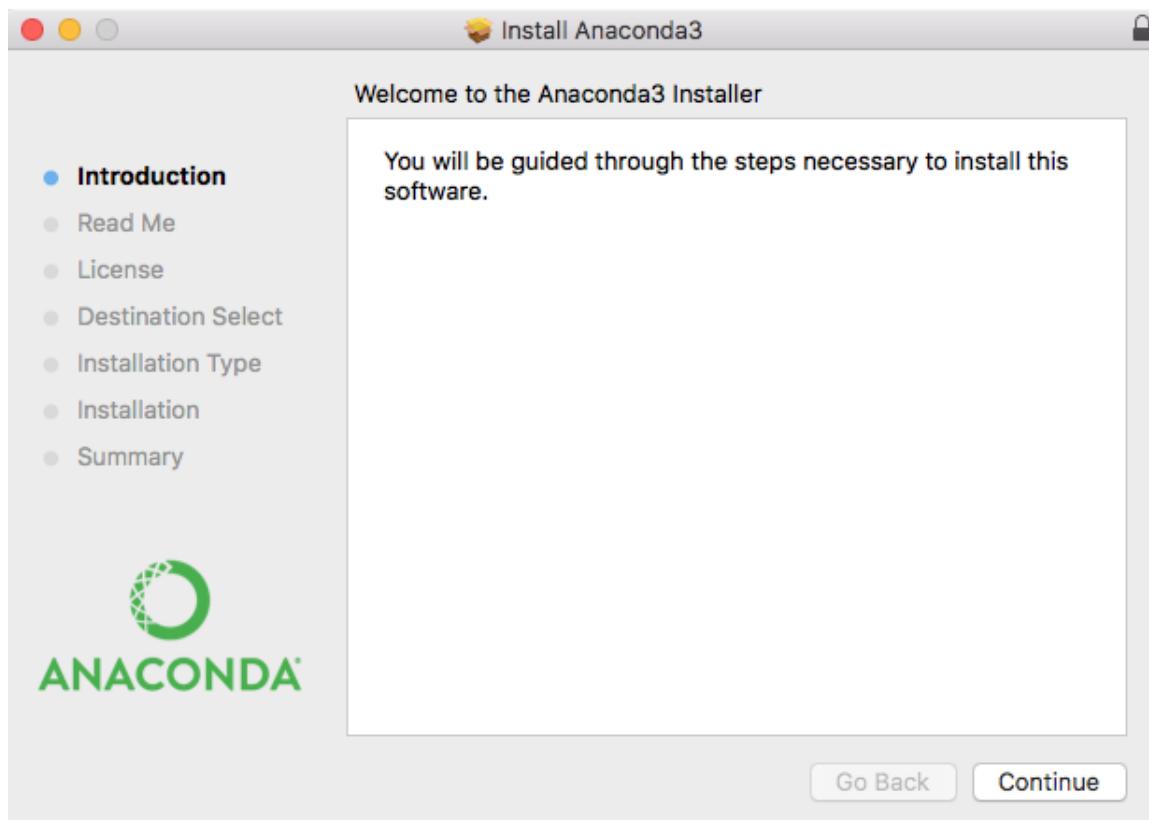


Figure B.3: Anaconda Python Installation Wizard.

Installation is quick and painless. There should be no tricky questions or sticking points.

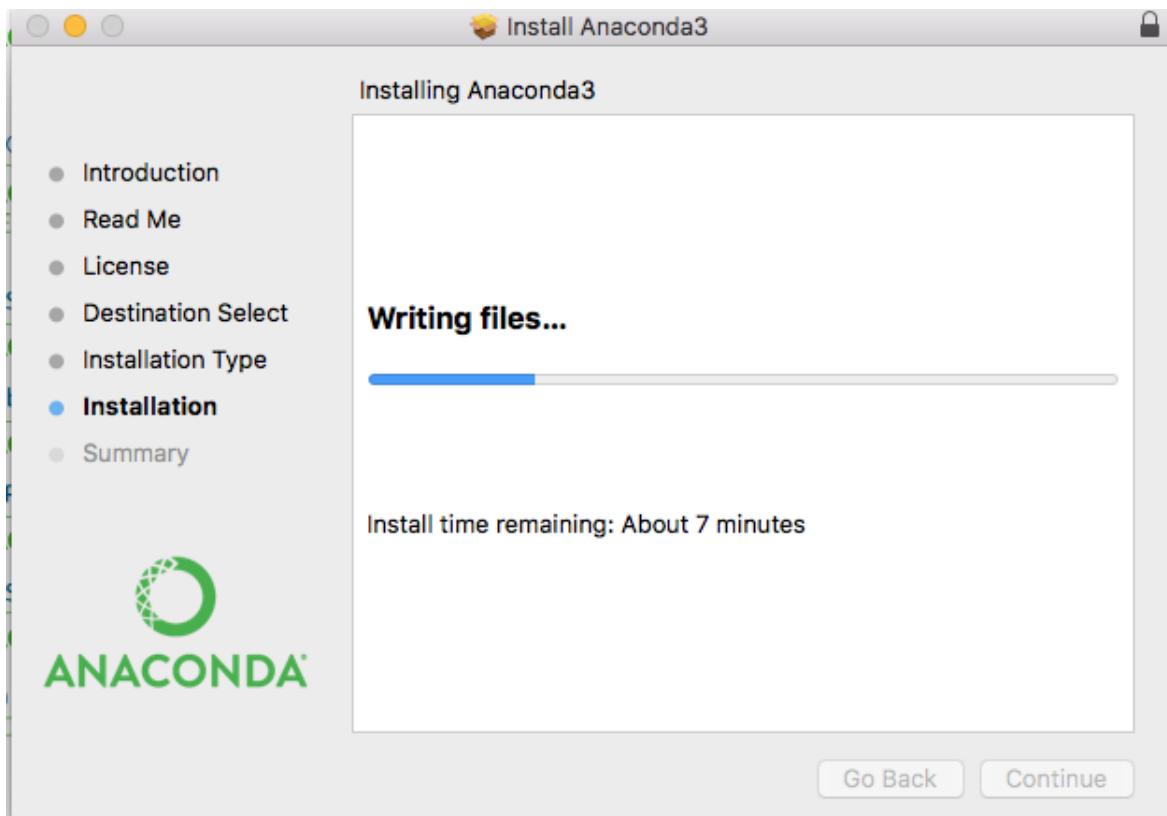


Figure B.4: Anaconda Python Installation Wizard Writing Files.

The installation should take less than 10 minutes and take up a little more than 1 GB of space on your hard drive.

B.4 Start and Update Anaconda

In this step, we will confirm that your Anaconda Python environment is up to date. Anaconda comes with a suite of graphical tools called Anaconda Navigator. You can start Anaconda Navigator by opening it from your application launcher.

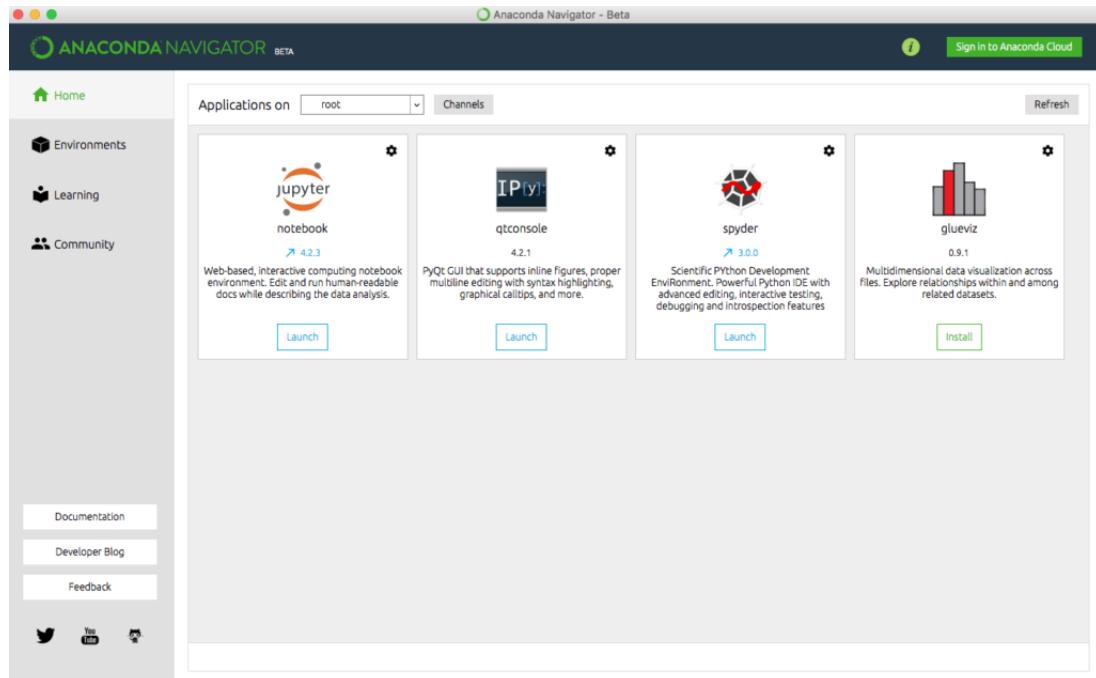


Figure B.5: Anaconda Navigator GUI.

You can use the Anaconda Navigator and graphical development environments later; for now, I recommend starting with the Anaconda command line environment called conda. Conda is fast, simple, it's hard for error messages to hide, and you can quickly confirm your environment is installed and working correctly.

- 1. Open a terminal (command line window).
- 2. Confirm conda is installed correctly, by typing:

```
conda -v
```

Listing B.2: Check the conda version.

You should see the following (or something similar):

```
conda 4.3.21
```

Listing B.3: Example conda version.

- 3. Confirm Python is installed correctly by typing:

```
python -v
```

Listing B.4: Check the Python version.

You should see the following (or something similar):

```
Python 3.6.1 :: Anaconda 4.4.0 (x86_64)
```

Listing B.5: Example Python version.

If the commands do not work or have an error, please check the documentation for help for your platform. See some of the resources in the *Further Reading* section.

- 4. Confirm your conda environment is up-to-date, type:

```
conda update conda
conda update anaconda
```

Listing B.6: Update conda and anaconda.

You may need to install some packages and confirm the updates.

- 5. Confirm your SciPy environment.

The script below will print the version number of the key SciPy libraries you require for machine learning development, specifically: SciPy, NumPy, Matplotlib, Pandas, Statsmodels, and Scikit-learn. You can type `python` and type the commands in directly. Alternatively, I recommend opening a text editor and copy-pasting the script into your editor.

```
# check library version numbers
# scipy
import scipy
print('scipy: %s' % scipy.__version__)
# numpy
import numpy
print('numpy: %s' % numpy.__version__)
# matplotlib
import matplotlib
print('matplotlib: %s' % matplotlib.__version__)
# pandas
import pandas
print('pandas: %s' % pandas.__version__)
# statsmodels
import statsmodels
print('statsmodels: %s' % statsmodels.__version__)
# scikit-learn
import sklearn
print('sklearn: %s' % sklearn.__version__)
```

Listing B.7: Code to check that key Python libraries are installed.

Save the script as a file with the name: `versions.py`. On the command line, change your directory to where you saved the script and type:

```
python versions.py
```

Listing B.8: Run the script from the command line.

You should see output like the following:

```
scipy: 1.4.1
numpy: 1.18.0
matplotlib: 3.1.2
pandas: 0.25.3
statsmodels: 0.10.2
sklearn: 0.22.1
```

Listing B.9: Sample output of versions script.

B.5 Install the Imbalanced-Learn Library

The imbalanced-learn library is an open source Python library that provides tools for working with imbalanced classification problems. The imbalanced-learn Python library, which can be installed via conda as follows:

```
conda install -c conda-forge imbalanced-learn
```

Listing B.10: Command for installing the imbalanced-learn library via conda.

Alternatively, you may choose to install using pip, as follows:

```
pip install imbalanced-learn
```

Listing B.11: Command for installing the imbalanced-learn library via pip.

You can confirm that the installation was successful by printing the version of the installed library:

```
# check version number
import imblearn
print(imblearn.__version__)
```

Listing B.12: Code to check that the imbalanced-learn library is installed correctly.

Save the script to a file `imbalanced_version.py`. Run the script by typing:

```
python imbalanced_version.py
```

Listing B.13: Run script from the command line.

You should see output like:

```
0.6.1
```

Listing B.14: Sample output from checking that the imbalanced-learn library is installed correctly.

B.6 Install the Deep Learning Libraries

In this step, we will install Python libraries used for deep learning, specifically: TensorFlow and Keras. Install the TensorFlow deep learning library by typing:

```
conda install -c conda-forge tensorflow
```

Listing B.15: Install TensorFlow with conda.

Alternatively, you may choose to install TensorFlow using pip, as follows:

```
pip install tensorflow
```

Listing B.16: Install TensorFlow with pip.

Install Keras with conda by typing:

```
conda install -c conda-forge keras
```

Listing B.17: Install Keras with conda.

Alternatively, you may choose to install Keras using pip, as follows:

```
pip install keras
```

Listing B.18: Install Keras with pip.

Confirm your deep learning environment is installed and working correctly. Create a script that prints the version numbers of each library, as we did before for the SciPy environment.

```
# check deep learning version numbers
# tensorflow
import tensorflow
print('tensorflow: %s' % tensorflow.__version__)
# keras
import keras
print('keras: %s' % keras.__version__)
```

Listing B.19: Code to check that key deep learning libraries are installed.

Save the script to a file `deep_versions.py`. Run the script by typing:

```
python deep_versions.py
```

Listing B.20: Run script from the command line.

You should see output like:

```
tensorflow: 2.0.0
keras: 2.3.1
```

Listing B.21: Sample output of the deep learning versions script.

B.7 Install the XGBoost Library

The XGBoost library is an open source Python library that provides an efficient implementation of the stochastic gradient boosting algorithm. The XGBoost Python library, which can be installed via conda as follows:

```
conda install -c conda-forge xgboost
```

Listing B.22: Command for installing the XGBoost library via conda.

Alternatively, you may choose to install using pip, as follows:

```
pip install xgboost
```

Listing B.23: Command for installing the XGBoost library via pip.

You can confirm that the installation was successful by printing the version of the installed library:

```
# check version number
import xgboost
print(xgboost.__version__)
```

Listing B.24: Code to check that the XGBoost library is installed correctly.

Save the script to a file `xgboost_version.py`. Run the script by typing:

```
python xgboost_version.py
```

Listing B.25: Run script from the command line.

You should see output like:

```
0.90
```

Listing B.26: Sample output from checking that the XGBoost library is installed correctly.

B.8 Further Reading

This section provides resources if you want to know more about Anaconda.

- Anaconda homepage.
<https://www.continuum.io/>
- Anaconda Navigator.
<https://docs.continuum.io/anaconda/navigator.html>
- The conda command line tool.
<http://conda.pydata.org/docs/index.html>
- Imbalanced-Learn Installation.
<https://imbalanced-learn.org/stable/install.html>
- Instructions for installing TensorFlow in Anaconda.
https://www.tensorflow.org/get_started/os_setup#anaconda_installation

B.9 Summary

Congratulations, you now have a working Python development environment for machine learning. You can now learn and practice machine learning on your workstation.

Part IX

Conclusions

How Far You Have Come

You made it. Well done. Take a moment and look back at how far you have come. You now know:

- The challenge and intuitions for imbalanced classification datasets.
- How to choose an appropriate performance metric for evaluating models for imbalanced classification.
- How to appropriately stratify an imbalanced dataset when splitting into train and test sets and when using k -fold cross-validation.
- How to use data sampling algorithms like SMOTE to transform the training dataset for an imbalanced dataset when fitting a range of standard machine learning models.
- How algorithms from the field of cost-sensitive learning can be used for imbalanced classification.
- How to use modified versions of standard algorithms like SVM and decision trees to take the class weighting into account.
- How to tune the threshold when interpreting predicted probabilities as class labels.
- How to calibrate probabilities predicted by nonlinear algorithms that are not fit using a probabilistic framework.
- How to use algorithms from the field of outlier detection and anomaly detection for imbalanced classification.
- How to use modified ensemble algorithms that have been modified to take the class distribution into account during training.
- How to systematically work through an imbalanced classification predictive modeling project.

Don't make light of this. You have come a long way in a short amount of time. You have developed the important and valuable foundational skills for imbalanced classification. The sky's the limit.

Thank You!

I want to take a moment and sincerely thank you for letting me help you start your journey with imbalanced classification for machine learning. I hope you keep learning and have fun as you continue to master machine learning.

Jason Brownlee
2021